

DRI File Copy

ESD-TR-72-308

ESD ACCESSION LIST

DRI Call No. 78369

Copy No. 1 of 2 cys.

MULTI-PATH CONTROL STRUCTURES
FOR PROGRAMMING LANGUAGES



Charles J. Prenner

August 1972

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Approved for public release;
distribution unlimited.

ESD RECORD COPY
RETURN TO
SCIENTIFIC & TECHNICAL INFORMATION DIVISION
(DRI) Building 1435

(Prepared under Contract No. FI9628-7I-C-0173 by Harvard University,
Cambridge, Massachusetts.)

AD758203

LEGAL NOTICE

When U.S. Government drawings, specifications or other data are used for any purpose other than a definitely related government procurement operation, the government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

OTHER NOTICES

Do not return this copy. Retain or destroy.

MULTI-PATH CONTROL STRUCTURES
FOR PROGRAMMING LANGUAGES

Charles J. Prenner

August 1972

DEPUTY FOR COMMAND AND MANAGEMENT SYSTEMS
HQ ELECTRONIC SYSTEMS DIVISION (AFSC)
L. G. Hanscom Field, Bedford, Massachusetts 01730

Approved for public release;
distribution unlimited.

(Prepared under Contract No. F19628-71-C-0173 by Harvard University,
Cambridge, Massachusetts.)



FOREWORD

This report was prepared in support of Project 2801, Task 280102 by Harvard University, Cambridge, Massachusetts under Contract F19628-71-C-0173, monitored by Capt Triston J. Rosenberger, ESD/MCIT, and was submitted September 1972.

This technical report has been reviewed and is approved.

Sylvia R. Mayer

SYLVIA R. MAYER
Project Officer

M. B. Emons

MELVIN B. EMMONS, Col, USAF
Director, Information Sys. Tech.
Deputy for Command & Mgmt Sys.

ABSTRACT

This dissertation applies the techniques of extensible languages to the problem of introducing multi-path control structures into programming languages. A control extension facility is defined which consists of a set of control primitives and a framework for combining them. The primitives are embedded in an existing extensible language--ELL. Using the facility, it is possible to realize both conventional and non-conventional control regimes by extension. Such extensions are simplified through the use of the control interpreter, which allows the programmer direct control over the assignment of processors to paths. A set of examples is presented which demonstrates the power of the facility for both the implementation and clarification of complex control structures.

Although the use of the primitives in the synthesis of control structures is emphasized, the primitives are also given a formal semantic definition which is used to demonstrate that they are feasible (i.e., they can be implemented on contemporary hardware) and that they have an efficient realization.

TABLE OF CONTENTS

	<u>Page</u>
List of Figures	viii
Synopsis	ix

Chapter 1. INTRODUCTION

1. Multi-Path Control Structures	1-1
1.1 Motivation	1-1
1.2 Design Criteria	1-7
1.3 Overview	1-10
2. Survey of Previous Work	1-16
2.1 Linguistic Work	1-16
2.2 Formal Specifications	1-24
2.3 Operating Systems	1-31
2.4 Other Work	1-33

Chapter 2. INFORMAL DESCRIPTION OF MPEL1

1. Processors	2-3
1.1 Interpreter as Processor	2-3
1.2 Multiplexing of Interpreters	2-6
1.3 Paths of Control	2-8
2. Paths	2-11
2.1 Informal Description of a Path	2-11
2.2 Path Creation and Deletion	2-15
2.3 Path Initialization	2-16
2.4 Path Evaluation	2-22
2.5 Data Sharing	2-24
2.6 Path Termination	2-27
2.7 Path Synchronization	2-29
2.8 Path Dependency	2-32
2.9 Intra-Path Control Primitives	2-37
3. The Control Interpreter	2-41
3.1 Communication with the CI	2-41
3.2 Synchronization	2-45
3.3 The Environment of the Control Interpreter	2-47
3.4 Path Scheduling	2-50

4. User Defined Scheduling	2-54
4.1 Scheduler Extension	2-54
4.2 Canonical Inactive Sets	2-61
4.3 Scheduling Errors	2-64
5. External Interrupts	2-66
5.1 Classes of Interrupts	2-66
5.2 Interrupt Structure	2-68
5.3 Processor Level Interrupts	2-70
5.4 Path Level Interrupts	2-76
5.5 Relation to Processor Multiplexing	2-82
5.6 Data Passage	2-86
6. Index to Chapter 2	2-89

Chapter 3. EXTENSIONS

1. Coroutines	3-2
2. Synchronization	3-6
3. Parallel Processing	3-10
4. Simulation	3-24
5. Monitoring and Relative Continuity	3-36
6. Backtracking	3-49

Chapter 4. THE FORMAL DEFINITION OF MPOL1

1. Introduction	4-1
1.1 Representation	4-1
1.2 Evaluator Recursion	4-4
1.3 Stacks	4-8
1.4 Synchronization	4-11
2. The EL1 Evaluator	4-14
2.1 Declarations and Initialization	4-15
2.2 Form	4-21
2.3 List Structure	4-23
2.4 Literal Procedure	4-24
2.5 Block	4-24
2.6 Declaration	4-26
2.7 Conditional	4-29
2.8 Selection	4-31
2.9 Assignment	4-34
2.10 Iteration	4-35
2.11 Procedure Application	4-39
2.12 Labelled Statement	4-46

3. The Control Primitives	4-47
3.1 GETPATH	4-48
3.2 PAP, PAPQ, DPAP, DPAPQ	4-50
3.3 PFETCH, PSTORE	4-54
3.4 TSET, CLEAR	4-56
3.5 MDEP, DEPEND	4-57
3.6 DELETE\PATH	4-64
3.7 GOTO, RETFROM	4-67
3.8 MYPATH	4-70
3.9 EVAL	4-71
3.10 COPY	4-72
3.11 CIA, CONTPATH	4-74
3.12 ENABLE\PRO, DISABLE\PRO, LEVEL, INUSE	4-79
3.13 ENABLE\PATH, DISABLE\PATH	4-82
3.14 MASK, UNMASK, INTERRUPT	4-85
3.15 STOP\PATH	4-87
4. Auxiliary Procedures	4-94
5. Primitive Procedures	4-100
6. Index to Chapter 4	4-110

Chapter 5. EVALUATION AND CONCLUSIONS

1. Other Facilities	5-1
1.1 Extended CIA Call	5-1
1.2 Extended Mode Facility	5-4
1.3 Termination of Dependents	5-9
2. Implementation Issues	5-12
2.1 Storage Management	5-12
2.2 Input-Output	5-20
2.3 Relation to an Operating System	5-21
3. Critical Discussion	5-25
3.1 The Control Primitives	5-25
3.2 The Formal Definition	5-33
4. Conclusions and Suggestions for Future Research	5-43

Appendix 1 Introduction to EL1

Appendix 2 Syntax of EL1

Appendix 3 CI Procedures and Interrupt Response Forms

References

LIST OF FIGURES

<u>Figure</u>		<u>Page</u>
2-1	Paths Q and P1 Before PAP	2-20
2-2	Paths Q and P1 After PAP	2-20
2-3	Nesting of Schedulers	2-59
2-4	The Definition of INITD	2-63
3-1	Trees x, y, and Modified y	3-4

SYNOPSIS

This dissertation applies the techniques of extensible languages to the problem of introducing multi-path control structures into programming languages. A control extension facility is defined which consists of a set of control primitives and a framework for combining them. Using this facility, it is possible to realize both conventional and non-conventional control regimes by extension. Such extensions are simplified through the use of the control interpreter, which allows the programmer direct control over the assignment of processors to paths. The use of the primitives in the synthesis of control structures is emphasized. However, the primitives are also given a formal semantic definition which is used to demonstrate that they are feasible, (i.e. they can be implemented on contemporary hardware) and that they have an efficient realization.

Chapter 1 gives the motivation for this research and contains a survey of related work.

Chapter 2 presents an informal description of the multi-path control facility. The primitives are embedded in an existing extensible language, namely, EL1 [Weg70]. We use the term MPEL1 to describe the language obtained through the addition of the control primitives to EL1. All of the material in this chapter — the control primitives and the

framework provided by the control interpreter — is original except for the intra-path control primitives EVAL, GOTO and RETFROM, all of which have counterparts in existing languages.

Chapter 3 describes how a variety of multi-path control structures can be defined as extensions to MPEL1. Although many of the examples have appeared in the literature, their straightforward realization in terms of the primitives and framework of MPEL1 is original.

Chapter 4 presents a formal semantic description of MPEL1. The definition is divided into two parts. First, a formal description of an EL1 evaluator is presented. It is similar to the definition of EL1 given in [Weg70], but has been updated to reflect changes in the language which have been included in a current implementation [Weg72]. The second part is a formal definition of the control primitives. This latter part and the modifications made to the semantic model in order to host evaluator multiplexing are original.

Chapter 5 contains some concluding remarks about the multi-path facility. First, a number of implementation issues are discussed. Second, an assessment of the control primitives and their formal model is given. Lastly, a number of areas for future research are described.

Appendix 1 presents a brief description of EL1.

Appendix 2 gives an augmented syntax for EL1. It is reprinted from [Weg70].

Appendix 3 presents the MPEL1 definitions of the control interpreter procedures and interrupt response forms described in chapter 2. All of this material is original

A brief description of this research was presented at the International Symposium on Extensible Languages, Grenoble, France, September, 1971, under the title "The Control Structure Facilities of ECL." A copy of the paper appears in the Symposium's Proceedings [Sch71].

Chapter 1

INTRODUCTION

1. MULTI-PATH CONTROL STRUCTURES

1.1 Motivation

A considerable amount of programming language research has been directed towards the development of extensible languages. The term 'extensible' has been applied to a number of quite different languages, and there is still disagreement in the field as to what characterizes a truly extensible language [Sch71]. Most 'extensible' languages have provided mechanisms for extension in one or more of the following areas.

- (1) Data type extension allows new data types to be created in terms of built-in or previously defined ones. It is usually possible to construct data types for arrays of homogeneous objects and structures composed of heterogeneous objects.
- (2) Operator extension allows for the definition of new operations or the redefinition of existing ones. For example, the meaning of '+' can be changed to cover addition over new data types.
- (3) Syntax extension allows the programmer to state his

algorithm in a more convenient notation than that of the basic language, provided that he can describe the mapping between the new notation and existing language constructs.

In each of the cases above, a language component, which had previously been a constant, becomes a variable. For example, in early high-level languages (ALGOL-60, FORTRAN) the number of data types available is constant - integers, reals, or n -dimensional arrays of integers or reals (where n is fixed at the time of compilation.) In an extensible language, the number of data types available is potentially infinite. The methodology of extensible languages has been to abstract what is fundamental in a given language component and then add to the language the primitives and framework necessary to allow the component to sustain variation.

The development of extensible languages may be considered a reaction to two other trends in programming languages. The first of these is the development of shell languages [Ch68]. These languages purport to service a wide class of users by making the language a conglomeration of the facilities needed by each class. The second trend is the development of specialized extensions to existing languages - the addition of SNOBOL-like pattern matching to ALGOL, for example. The former trend is not viable since the shell becomes quite unwieldy as the number of

application areas increase. The latter simply adds more dialects to Babel.

One of the most popular examples of the second trend is the addition of multi-path control operations to existing languages. [An65][Op65][Da66]. Here we have proposals for extensions to allow for asynchronous tasks, coroutines, fork-join constructions, synchronization operations, simulation primitives, and the like. Numerous papers have appeared in the literature which describe how one or more of the above can be added to some language (usually ALGOL-60.) Unfortunately, most of these proposals are incomplete, usually taking the form of an English language description or a sketch of an implementation. An explication of the effect of extension upon the language as a whole or a study of what fundamental operations underlie all of these extensions are never presented.

Extension facilities to allow for multiple paths of control have been ignored in most extensible language proposals. ^{*} This is surprising since the number of (ad-hoc) extensions which have been proposed make this area ripe for the application of the techniques of extensibility.

This thesis attacks the problem of introducing multi-path control structures into programming languages

*

The few exceptions are discussed in section 1.2.1.

through the use of the extensible language method, namely, a set of language primitives and a framework are proposed which allow for the synthesis of all known multi-path control structures and, hopefully, for the synthesis of an unspecified number of others. The primitives allow for systematic variation in four areas.

- (1) Path Organization - Paths of control (sequential processes) do not have to be designated as conforming to any particular control behavior (such as a task or coroutine structure.) The control relationship between paths is determined entirely by their use. Facilities for data sharing are provided commensurate with the generality of the control discipline in effect.
- (2) Scheduling - Any multi-path facility must surely allow for the concurrent activation of paths, i.e. parallel processing. If the number of paths to be activated concurrently exceeds the number of processors available, then some path-scheduling technique must be employed. The proposed framework allows the scheduler to be defined at the language level.
- (3) Synchronization - Whenever a language admits concurrent evaluation, some mechanism must be provided to allow the parallel paths to synchronize their activities. Although no synchronization

operator or special data type is assumed,^{*} the necessary handles are provided so that hand-tailored synchronization operations may be constructed.

- (4) Interrupts - An interrupt facility is provided which allows a path to be interrupted by a signal which is generated by another path or by an external source.

The primitives are defined as extensions to the evaluator of an existing extensible language - EL1 [Weg70]. EL1 was chosen as the host language for a number of reasons. First, EL1 contains no multi-path facilities. Second, EL1 has both stack and dynamic storage allocation. The latter provides a convenient mechanism for data sharing between paths.^{**} Third, the 'dynamic' structure of EL1 provides an environment in which the evaluation of a program is not tied to its textual structure. Fourth, an implementation-oriented formal definition of EL1 exists. Hence, the effect of proposed control primitives upon the language can be determined by performing modifications to the semantic model. Finally, embedding the primitives in EL1 avoided the creation of yet another extensible language.

*

A TEST-AND-SET operation is assumed in order to allow the control primitives to synchronize their activities, c.f. 1.1.3.

**

Here we refer to the fact that EL1 uses a dynamic scope rule to identify the meanings of free variables (as in LISP) and to the fact that variables may be bound to fragments of EL1 programs (called FORMs) which may be evaluated in any environment.

The multi-path primitives are described both in English and in terms of a revised formal model of EL1. The former serves as an informal introduction and the latter provides a precise definition of the semantics of the primitives and their relation to the EL1 evaluator. The formal definition is crucial for two reasons. First, most of the control primitives do not have counterparts in conventional programming languages. Thus, the informal description must concentrate on motivating the primitives and giving a general description of their actions. An attempt at completeness in this section would make it essentially unreadable. Second, to propose sophisticated linguistic primitives without giving a model is a relatively worthless pursuit - only the language designer will ever understand exactly how they work.

For the remainder of the thesis we will use the term 'MPEL1' (multi-path EL1) to denote the language obtained by adding the control primitives to EL1. The term 'EL1' will be used to refer to the original definition of EL1.

1.2 Design Criteria

In this section we will discuss a number of criteria which might be used to judge the merit of a set of multi-path primitives and their formal model. In chapter 5, we will evaluate the MPFL1 primitives and model in terms of these criteria.

The first criterion is 'cost.' There are a number of measures which apply. First, the amount of 'code' needed to implement the primitives should be small. Hopefully, the primitives will use facilities which already exist in the language whenever possible. Second, the primitives should be reasonably efficient. If they are too slow, they will merely be curiosities to be played with instead of tools to be used to solve real problems. Lastly, and most importantly, the overheads associated with the multi-path facility should not be distributed throughout the language evaluator; a program which does not use the primitives should not pay for their existence in the language.

The next criterion is the generality of the universe in which the primitives are defined. Here, we refer to the number of processors which are available for the simultaneous evaluation of paths. Any set of primitives which are defined in a multi-processor environment will surely be acceptable if the number of processors is restricted to one. The converse, however, is not

necessarily true. Primitives which are feasible in a uni-processor environment may prove to be quite expensive in the more general environment. Secondly, problems which are non-issues with a single processor become significant in light of multi-processing. For example, with a single processor only one primitive can be executing at any given time. Hence, the primitives do not require any explicit synchronization. With multi-processing, two primitives may execute simultaneously and therefore may require synchronization. Thus, a multi-processing environment exposes issues which do not arise in the restricted single processor case.

Turning to the formal model, the most important criterion is that the model should explain the primitives. Presumably, the language level control primitives are designed to facilitate the synthesis of multi-path control structures. They are cast at a high enough level to suppress the constant and display the variable. Hence, they are probably sufficiently complex that their feasibility or implementation is not immediately obvious. The formal definition should explicate how the language primitives may be constructed from some smaller set of primitives which are intuitively acceptable, i.e. they can be implemented on existing hardware.

We close this section by posing the question 'Why

bother?', i.e. are multiple paths of control really necessary in programming languages? Formally, of course, the answer is no - multiple paths of control do not add any computational power. In this sense, data definition facilities are just as useless. In both cases, however, their value lies in the representational power provided. Data type extension facilities allow one to define the data structures which are appropriate for a given problem. The resulting algorithm is usually cleaner and more concise than one in which the data is represented in some indirect fashion using some fixed set of data types. Similarly, algorithms which call for a multi-path structure suffer greatly when they are compressed into a single path of control. In some sense, the situation is worse for control than for data. To illustrate, it is usually possible to use one data structure to represent another with a major loss of notational convenience but a tolerable loss of speed. For example, arrays of integers can be used to represent lists. However, in order to simulate a multi-path control structure with a single control path one may have to construct an interpreter. In this case, the result will certainly be intolerably slow!

1.3 Overview

In this section we present an outline of the multi-path facility. In all cases, the topics are covered in more detail in the chapters that follow. This section is included in the hope that an initial pass through the facility will aid the reader in his understanding of the components as they are presented linearly in the sequel.

The first topic to be considered is the underlying machine model. Since the language is to be machine independent, no specific machine organization is assumed. However, a number of features that can be found in most contemporary hardware systems are presumed. It is assumed that there exist a fixed number of processors available for the simultaneous evaluation of paths. Each one may evaluate only one path of control at a time. A processor must always be kept busy. Thus, an idling path is defined for each one - the path it evaluates in the absence of any 'real' paths. Associated with each processor are a set of external interrupts, e.g. timer interrupt, light-pen interrupt, or processor to processor interrupt, and a priority interrupt system. Processor communication and synchronization is achieved through the interrupt system and through the use of an interlock instruction which relies upon the arbiting property of memory (TEST-AND-SET).

A path records the dynamic evaluation of an EL1 program. Associated with each path is an environment which contains the name-value bindings created by the dynamic execution of the program and an activation record (an EL1 structure) which contains a handle on the path's environment and information that describes the status of the path. Activation records also provide a linguistic means of talking about paths in the language, e.g. most control primitives take as argument a pointer to the activation record of the path to which the primitive is to be applied. Control primitives are defined which allow for path creation and deletion, modification of a path's environment, initialization of a program to be evaluated in the path, interruption of one path by another, transfer of control between paths, etc.

A path is active if a processor is currently evaluating a program in the path's environment. Since the evaluation of a program in a path is isomorphic to the data structure modifications made in the path by the processor, we may speak of the path itself as being evaluated by the processor. If a path (P) is being evaluated by a processor (Q), then we say that Q is assigned to P. A path is not active if a processor is not currently assigned to it. A path is being modified if it is active or if a control primitive that affects its environment is being applied to it.

A path may only be modified by one processor at any given time. If a path is active, then the path is being modified by the processor which is evaluating it. If an environment modifying control primitive is being applied to a path, then it is being modified by the processor of the path which has executed the primitive. A processor TEST-AND-SETs a memory location in the activation record of the path to be modified. If two processors simultaneously* attempt to modify a path, then a runtime error results. Thus, the TEST-AND-SET instruction is used only to insure that two (or more) processors do not modify a path simultaneously, i.e. to protect the system against fatal language-level program bugs.

The framework in which the primitives are cast completes the built-in multi-path facility. Essentially, the framework consists of the existence of a distinguished path, the control interpreter (CI), which is treated specially by the control primitives. It is the only path to which other paths may pass control. This is achieved by means of the control primitive CIA which transfers control to the CI, specifying a function to be applied in its environment. The CI path, in conjunction with the control primitives and the interrupt facility, provides the handle

*

If a processor TEST-AND-SETs the word and finds it 'unlocked', then it simply continues the evaluation. If, however, the processor finds the word 'locked', then it generates the runtime error.

necessary for the synthesis of multi-path control structures. Two properties of the control interpreter path facilitate such constructions. First, the execution of any function (passed via CIA) in its environment is indivisible with respect to CIA calls of the same function by other paths, i.e. if two paths simultaneously CIA some procedure, say *f*, then the execution of one CIA call of *f* will run to completion before the other is allowed to begin. The two executions of *f* are ordered linearly in time. Second, control transfers between paths must go through the control interpreter. Thus, it acts as a control switchyard. The consolidation of indivisibility and path-switching in the control interpreter simplifies the synthesis of control structures without any loss of descriptive power. For example, the control interpreter can be used to realize any sort of synchronization operation or path scheduling regime by extension.

It is important to distinguish the control primitives and the CI path from the program being evaluated in the control interpreter's environment. This program, written in EL1, uses the fact that it is executed in the CI path in conjunction with certain control primitives and the interrupt system to provide an initial extension to the built-in facility. In particular, the program evaluates CIA called procedures and provides a simple path scheduler which allows for the synthesis of concurrent processes. Since

this program is written in EL1, it is easily understood and is available for modification or redefinition by the user.

The CI framework, in conjunction with the control primitives, allows for the construction of conventional multi-path organizations such as coroutines and concurrent processes. In addition, MPEL1 can host non-conventional control regimes such as monitoring and relatively continuous evaluation. Chapter 3 gives examples which demonstrate how these control structures, among others, may be realized as straightforward extensions in MPEL1.

To summarize, MPEL1 provides an extensible multi-path facility. The extensible nature of MPEL1 is best viewed in terms of three concentric levels:

- (1) The control primitives and the existence of the CI are built-in and constitute the basis for the multi-path facility.
- (2) The MPEL1 program which is evaluated in the CI environment executes CIA called procedures and performs path scheduling.
- (3) MPEL1 programs utilize the control primitives and communicate with the CI in order to produce a given control regime.

We conclude this section with a brief comment about the formal model. In the previous section, we indicated that it is desirable to have the formal model explain the control

primitives. The meta-language used to describe MPFL1 is EL1 with the inclusion of only four control primitives (TSET, CLEAR, EVAL, GOTO.) Thus, the semantics of the primitives are specified in terms of EL1 and a small set of control primitives. Since the primitives used in the meta-language have a straightforward realization on existing hardware, the model is pragmatically valid at the base level, c.f. 5.3.2.

2. SURVEY OF PREVIOUS WORK

This dissertation builds on previous work in a number of programming language research areas, namely, introduction of multi-path facilities, extensible languages and formal semantic specification. In addition, our research also touches upon work done in operating systems and abstract models of parallel systems. A complete survey of all of these areas would surely be beyond the scope of this paper. Hence, we will restrict our discussion to those papers which are directly relevant to the current work. Comprehensive bibliographies may be found in [ACM70][Ste66][Chris69].

2.1 Linguistic Work

Most proposals for language additions which allow for the creation of multiple paths of control have been attempts to permit user specification of program segments that may be executed concurrently. Some sort of synchronization facility is usually provided to allow the parallel segments to coordinate their activities.

Anderson [An65] proposes additions to ALGOL-60 to provide for parallel processing. He introduces five statement types: fork, join, release and terminate. The fork statement specifies a list of labels to which control is to be passed in parallel. The last logical statement in

the body of code following each label is to be either a goto X, where X is the label of a join statement or a terminate statement, which indicates that this path has no successor. The join statement specifies the labels of the parallel paths that must complete before control can pass through the join. Obtain and release are used to provide synchronization. The obtain statement prevents other paths from accessing the values of the variables in the obtain's variable list. Release is the logical counterpart of obtain, i.e. it allows access to variables previously obtained.

Opler [Op65] suggests the addition of a DO TOGETHER statement to FORTRAN. The statement specifies a set of DO-loops which may be evaluated concurrently. When all paths have completed, processing continues with the statement following the DO TOGETHER.

Variations on the above have been proposed by Conway [Co63], Wirth [Wi66], who suggests the use of the operator and to indicate a lack of commitment in the sequencing of program segments, and Gosden [Go66], who recommends the use of and for paths that rejoin and also for ones that do not.

Dijkstra [Di68a] proposes the introduction of a parallel compound statement (parbegin parend) into

ALGOL-60, where the statements of the block are to be evaluated concurrently. Evaluation of the block is completed when all statement evaluations have completed. Synchronization is achieved through the use of semaphores and their associated operations, P and V.*

PL/I [Be70] allows procedures to be called as separate asynchronous tasks. The task structure, however, is strictly hierarchical - a created task is always dependent upon the block of the parent task that created it. If control returns from a creator block, then all tasks created by that block are forcibly terminated. Synchronization is achieved through the use of EVENT variables; one WAITS for an event to occur. The occurrence of an event is signaled by COMPLETION, or by the I/O subsystem - in the case of event variables associated with I/O activities. PL/I also provides an interrupt handling facility. The programmer may associate a program (called an 'on-unit') with a 'condition' which may be built-in (e.g. SUBSCRIPTRANGE) or user-defined. The on-unit is evaluated if the condition obtains during the evaluation of the program. User-defined conditions must be raised explicitly by means of a SIGNAL statement.

Most of the multi-path facilities described above have

*

See section 3.2 for a complete description of semaphores.

not been fully incorporated into their host languages. Hence, the semantic relation between the control primitives and the rest of the language is occasionally quite fuzzy. For example, in Anderson's proposal the effect of non-local gotos out of parallel segments is not explained. We note, however, that these proposals are suggestions of desirable language features and do not purport to be complete language designs.

Next to parallel processing, coroutines have been the form of multi-path control most frequently discussed in the literature [Co63a],[McIl],[Kn68]. Coroutines are useful whenever the solution to a problem cannot be easily cast into a single hierarchical structure. They have been characterized in many ways - from mutual subroutines that may call upon each other to procedures that use their own storage to retain information about their internal state between calls.

Neither of the above viewpoints is fruitful, since they both attempt to explain coroutines in terms of hierarchical control. For example, the use of own storage allows a procedure to construct its own separate mini-hierarchy so that upon subsequent calls it can resume execution from where it left off. A more reasonable view of coroutines is in terms of multiple-paths of control in which each path maintains its own control hierarchy. When one path wishes

to 'resume' a coroutine path, it simply transfers control to the path. Since the hierarchies are separate, the state of the original path remains intact.

Discrete simulation languages, such as SIMULA [Da66], use a multi-path coroutine structure to effect clock-driven simulations. Processes are maintained on a queue, termed the sequencing set (SQS), in the order in which they are to be evaluated in 'system time'. A number of processes may be set to be evaluated at the same 'system time', i.e. concurrently with respect to the system being simulated. However, these processes are evaluated in an interleaved manner as coroutines, not as parallel processes. Control resides in one process until it either terminates, reschedules itself for later evaluation, or passes control to another process. This mode of operation is called 'quasi-parallelism.' To achieve the effect of concurrent processing, the programmer must explicitly deal with the scheduling of processes.

SIMULA provides a large number of scheduling operations to facilitate management of the SQS; primitives exist which allow processes to be removed from the SQS, added to the SQS before or after some particular process, or added before all processes to be evaluated at a specified time. More recently, SIMULA67 [Da70] has recognized the essential coroutine structure of SIMULA and allows these scheduling

operations to be realized as extensions using the two operations DETACH (which passes control out of a coroutine process) and RESUME (which passes control to a coroutine process) and the ability to define a SQS in the language (via a data definition facility.) These operations also allow other multi-path control structures to be synthesized (in a uni-processor environment.)

In lieu of a survey of extensible languages, which would unnecessarily lengthen this section, we will limit ourselves to discussion of the multi-path control facilities of a number of extensible languages. For general surveys of extensible languages see [Chris69][Ger69][Sch71].

AICOL 68 [vanW69] allows for 'collateral elaboration' where the sequence in which a set of expressions are evaluated is left indeterminate, e.g. they may be evaluated either simultaneously, sequentially in any order, or in an interleaved fashion. AICOL 68 also allows parallel clauses in the spirit of Dijkstra's parallel compound statement, where

parbegin s1; s2; s3 parend

becomes

par(s1, s2, s3).

The constituent statements of a parallel clause are elaborated collaterally. The programmer may use semaphores (objects of mode sema) to synchronize the operation of the

statements. Here, the P and V operations appear as down and up, respectively. The monadic operator `'/'` is used to initialize a semaphore, i.e. `/` takes an int argument and returns a new sema whose integer count is initialized to the value of the int.

Standish [St68][St69] has proposed a number of control features for PPL, including mechanisms for parallel processing, interrupts, continuously evaluating expressions (a construct that allows the free variables in an expression to be monitored so that if the value of any one of them changes, the expression is immediately reevaluated,) and control contracts which allow the user to manipulate the control interfaces between processes.

More recently, Poupon [Po71] has implemented a number of these features in an experimental version of a current implementation of PPL [Ta71]. A PPL process is a data structure which represents the dynamic incarnation of a procedure call. The components of a process include the formal parameters and locals of the procedure, a STATUS component (which may take on the values ACT (active), SUS (suspended), or TED (terminated) and a RESULT component which is used to reference the 'result' of the process. The values of these components may be selected and modified by other processes, e.g. if P references a process then

```
P[STATUS] <- SUS
```


suspends the process P. All active PPL processes are evaluated concurrently.

PPL also contains the following two control operations. First, it is possible to specify that a PPL process is to be evaluated relatively continuous [Fi70] to all other processes. The evaluations of all other processes are delayed until the process terminates or indicates that it has completed the desired (relatively continuous) processing. Second, PPL provides a type of continuously evaluating expression in the WAITUNTIL statement, e.g.

WAITUNTIL(A + B = 3)

When a WAITUNTIL is encountered, the expression is evaluated. If the value is TRUE, then the process continues. Otherwise, the process is suspended with control positioned at the WAITUNTIL statement. It is made active whenever the value of any variable in the expression is changed. Hence, the process will continue as soon as the expression becomes TRUE.

OREGANO [Be71] allows for the construction of coroutines and parallel tasks (perhaps with associated priorities.) Synchronization is achieved through the use of event variables and the operations wait (wait for event to occur), cause (cause the event), and reset (re-initialize the event variable). Tasks, coroutines, and procedure calls are treated in a homogeneous fashion, namely, the invocation

of each involves the allocation of a contour which contains local variables, environmental information and an instruction pointer. Contours are managed using a retention strategy, i.e. a contour remains in the system as long as it is reachable from some 'active' contour. Contours which are no longer reachable are returned to the free storage pool by an automatic reclamation technique such as garbage collection. The retention strategy allows for a more flexible tasking structure than some of the languages described above, say PL/I, since the environment required by a created task will remain as long as necessary, independent of the actions of the creator task.

2.2 Formal Specifications

We now turn to a discussion of work done in the area of formal semantic models of programming languages. The approaches taken to this problem have been quite diverse, ranging from compiler-based specifications [Car66] to string processor models [vanW66]. Unfortunately, most of these models are oriented towards describing languages which admit only a single path of control. This is not surprising, however, since most languages have a single-path control structure - the notable exceptions being the ones described earlier in this section. For our purposes, it will suffice to review only those papers which are reasonably relevant to

this paper. For more complete surveys of the field see [Ste66][Wegn69].

Landin [Lan65][Lan65] has investigated the use of the lambda-calculus as a basis for the formal description of programming languages. He demonstrates how various language constructs can be cast as lambda-expressions and gives a mechanical procedure for the evaluation of lambda-expressions in terms of an interpreter for an automaton - the SECD machine. The applicative aspects of programming languages (recursion, parameter bindings, scope rules) are handled reasonably in this approach. However, the more imperative aspects of languages (assignment, transfers of control) must be modelled either by twisting them into applications or by introducing imperative features into the lambda-calculus.

McCarthy [McCar66] proposes a language definition method which uses a state vector to hold the current values of all variables accessible to a program. The result of evaluating a program P in language L with respect to an initial state vector V_0 , is defined to be the final state vector V' which is obtained by using a semantic function F/L associated with language L to sequence through P and produce state vectors $V_1, \dots, V_n = V'$ which record the successive values of the variables used in P . Hence, F/L acts as an interpreter for programs written in L .

ULD, the method and meta-language for language definition developed at the IBM Vienna Laboratories [Luc68a][Luc68b], must be considered the most ambitious effort in the field. The original work was undertaken to provide a formalism for the formal definition of PL/I. More recently, ULD has been successfully used to describe the semantics of other languages [Ger70][Rey69].

Basically, one describes the semantics of a language L by describing a basic abstract machine which is composed of a set of machine states and a (possibly non-deterministic) state transition function \wedge . Corresponding to any program P in L there exists an initial machine state S_0 . A computation is a sequence of states S_0, \dots, S_n , such that $S_{i+1} \in \wedge(S_i)$. A machine state is represented as a structured object, i.e. as a finite tree with named components. All of this is cast in a meta-language which is a conglomeration of conditional expressions, functional composition, the propositional calculus, and two operators (a selector and constructor) used to manipulate the structured objects.

McCarthy's formalism and ULD utilize two significant techniques. First, programs are represented abstractly as data structures which display the essential semantic structure of the program, while suppressing human-oriented syntactic sugaring. McCarthy defines the term abstract

*

syntax to describe this representation*. Second, the formalisms are interpreter-based, i.e. the semantics of a language are described by an interpreter (written in the meta-language) which evaluates abstract representations of programs in the semantic environment provided by the model. Hence, one language is defined by describing its semantic interpreter as a program in another.

A number of criticisms may be leveled at the two formalisms. First, the semantic environments in which the meta-languages are cast are unnecessarily restrictive. Here, we refer to the data structures of the meta-language which are used to record the state of the computation. McCarthy can use a simple fixed-length vector since the number of variables in any program in the language he is defining (a restricted subset of ALGOL-60) is constant and control can be described by a single statement number. In UID, although the tree structures provides a more flexible environment than a single state vector, restrictions on sharing of components force circumlocutions in the representation of common program language constructs. Second, although it contains standard language constructs, the UID meta-language uses an obscure notation in which familiar concepts are recast in unfamiliar settings. Hence,

*

Note that a complete language definition must include a specification of the concrete (written) representation of the language and a description of the mapping from concrete to abstract form.

learning ULD is as difficult an intellectual effort as understanding some of the languages it is used to describe.

Wegbreit [Weg70] resolves these issues, to some extent, in his formal definition of EL1. The language is defined by presenting a set of EL1 programs which constitute an EL1 evaluator. Hence, EL1 serves as its own meta-language. The data definition facility provides a sufficiently rich set of data structures so that the abstract syntax representation of programs and the semantic environment necessary for the evaluation of EL1 programs can be represented both directly and clearly. Because the direct representation of semantic structures and the fact that EL1 is a fluent notation for expressing algorithms, the formal definition is extremely readable. Complete understanding of the language is achieved by an iterative process, in which one's understanding of the formal definition reinforces one's understanding of the language, and conversely.

EL1 raises two related issues concerning formal definitions. The first of these is linguistic circularity. As Wegbreit notes, some such circularity is inescapable. To define a language L , one uses a meta-language L' . But how is L' defined?. Either $L=L'$ (as in the definition of EL1), L' is self evident and requires no formal definition, or L' is defined by yet another language L'' . The last of these choices yields a potentially infinite regress, unless the

chain is terminated by introducing a circularity or using a meta-language whose definition is obvious, e.g. a Turing machine representation. While simple meta-languages are logically attractive, they are inappropriate frameworks in which to cast language definitions. Either one becomes lost in the details associated with the simplistic language or one builds layers of definition on top of the language, in which case each layer must be examined for correctness. In addition, the evaluation process as represented in the simple language may misrepresent the essential qualities of the language mechanisms. This leads us to our second issue - implementation independence. Here, we do not refer to machine independence (i.e. non-reliance upon a specific machine organization) but rather to whether or not the formal definition should encompass a preferred data structure organization to be used in an implementation of the language. For example, ULD defines PL/I without indicating any possible implementation, whereas, the EL1 formal definition is cast in terms of a specific set of data structures to be used by the evaluator. We will return to this issue again in section 5.3.2.

One additional property of the EL1 formal definition must be discussed. The property may also be found in McCarthy's formalism and in classic definitions of LISP [McCar60]. Although in all of these formalisms the name-value environment in which a program is evaluated is

described explicitly (Wegbreit's name-pdl, McCarthy's state-vector, and LISP's A-list,) the control structure of the program is implicitly recorded in the recursive procedures of the semantic interpreter. This presents no problem in the formalisms described above since the languages defined allow for only a single path of control. However, if a language admits multiple paths of control, where each path can affect the intra-path control structure of another, then the program's control structure must be removed from the interpreter and included as part of the semantic environment so that the effects of these actions may be clearly explicated.

One formalism that attempts to include control structure as part of the semantic environment is Johnston's contour model [Jo71]. The model has been used in the design and specification of OREGANO [Be71]. The model consists of two components: a fixed reentrant algorithm and a time-variant record of execution. The latter is realized by nested contours which may be used to represent procedure or block activations. A processor is defined as an (ep,ip) pair, where ip is a pointer to an 'instruction' and ep is a pointer to a contour, which in turn defines the environment (by its relation to other contours) in which the instruction is to be executed. Many such processors can be defined and represent loci of control within the program. A fundamental axiom of the model is that contours are managed using a

retention strategy, c.f. 1.2.1. Languages defined using the model tend to exploit this axiom to the hilt as opposed to exploring other implementation strategies (e.g. stacks) which may be more efficient in certain cases.

2.3 Operating Systems

Computer operating systems have made use of the concept of multiple paths of control as a means of achieving a more efficient utilization of hardware and as a design methodology. In the former case, it has been observed that user programs (processes) do not require the use of a processor at various times during their execution, e.g. while waiting for I/O. Hence, it is profitable for the system to maintain more processes than processors and multiplex the processors across the processes as required. In the latter case, it has been found useful to describe an operating system as a society of cooperating sequential processes, associating one process with each user program and one process with each peripheral device [Di68b].

Saltzer [Sa66] defines the Traffic Controller as the program responsible for the orderly switching of processors between processes. A set of primitives are defined which allow a process to specify to the controller that (a) it has no further use for its processor, (b) it should be given a

processor again at some later time, (c) the further execution of some other process is to be stopped, and (d) some process can now make use of a processor. This model was incorporated into the MULTICS system [Cor65]. Rappaport [Ra68] discusses his experience with two implementations of the MULTICS Traffic Controller. In the first version many processes are allowed to execute inside the Traffic Controller simultaneously to prevent the Controller from becoming a system bottleneck. This requires numerous interlocks to insure that the processes do not interfere with one another. In the second version, only one process is allowed to execute inside the controller at any given time. Thus, only a single global interlock is required. Rappaport notes that both the size of the controller and the time required to execute the primitives were reduced significantly in the second version. Madnick [Ma68] shows that the use of a single global interlock will not cause a bottleneck unless the number of processors in the system is large, e.g. more than 5.

Wirth [Wi69] advocates the removal of input-output interrupts from machine language programming and suggests that they be replaced by a set of instructions which allow for the creation, termination, and synchronization (using semaphores) of parallel processes. In addition, a generic I/O instruction (DOIO) is proposed which performs a specified I/O activity to completion. The programmer can

conceptualize his program in terms of processes in which concurrent I/O is realized by starting a process to perform the I/O and then waiting (via a P operation) until the I/O is complete as opposed to fielding I/O interrupts at arbitrary points in his program. Wirth gives an implementation of the instructions as subroutines which are invoked by supervisor calls on an IBM 360.

2.4 Other Work

Before we conclude this survey, we must discuss three recent works in the area of control structures which do not fit conveniently into any of the research areas described above.

Leavenworth [Lea69] describes a language in which a programmer can define his own control structures since he can access the state of the language interpreter. The language, McG360 [Bur68] is similar to ISWIM [Lan66] and the interpreter resembles the SECD machine interpreter. It is possible to save entire machine states, construct new states, modify saved states, and install some saved state as the current machine state. For example, to simulate non-deterministic control using the primitives proposed by Floyd [Fl67] one saves one copy of the machine state for each value of the choice function, i.e. at each point at

which a non-deterministic choice must be made the machine state is replicated as many times as necessary. When a choice leads to failure, a saved state (which corresponds to a choice point) is installed as the current state. Coroutines can be obtained by defining a resume function which saves the current machine state for later resumption and restores some saved state. Unfortunately, most of the interesting control structures are obtained using the concept of 'saving entire machine state', which does not lend itself to an efficient implementation. In addition, concurrent operation is achieved by the multiplexing of a single processor (the interpreter) across machine states.

Fisher [Fi70] describes a set of control primitives which 'span our conceptual notion of control ... and can be easily composed to form more specialized control structures.' Six primitives, which are embedded in a programming language (CDL), are defined: seq which specifies that a set of statements are to be evaluated sequentially, par which specifies that a set of expressions are to be evaluated independently, cond which is similar to the LISP conditional [McCar60], monitor which allows an expression to be evaluated as soon as a condition becomes TRUE, synch which allows for the indivisible evaluation of an expression and cont which allows the evaluation of an expression to be relatively continuous with respect to the evaluation of

other control paths.* Fisher's return operation, which returns control from one process to another, may be considered a seventh primitive.

Fisher gives three definitions of the primitives: (a) in English, (b) a CDL interpreter written in CDL, (c) a CDL interpreter which uses only seq and cond. The first of these is useful as an informal description but, of course, is not precise. In the second, the more interesting primitives (monitor, synch, and cont) are defined by direct circularity (e.g. if a CDL program performs a cont, then the interpreter performs a cont.) Such circularity is acceptable (and unavoidable) in the formal definition of some language primitives. For example, in a definition of LISP, CAR, CDR and CONS are defined using CAR, CDR, and CONS directly. Here, the direct circularity is acceptable since the operations are intuitively clear and involve simple manipulations of well defined data structures. Fisher's primitives, however, involve complex actions performed upon less well defined structures, e.g. no clear definition of the term 'process' is given. Hence, their definition by direct circularity is suspect since it provides little insight into the mechanisms involved. In the last definition, pseudo-parallel processing is achieved by managing the processes on a queue and evaluating them (one

*

Conts may be nested. Hence, many levels of relative continuity may be invoked.

at a time) according to their level of relative continuity.

Thomas [Tho71] addresses the question 'How can processes be represented in order to facilitate synthesis of complex behavior patterns.' His answer is cast in terms of a state-oriented model in the spirit of Landin and Leavenworth. Here, each process has its own processor. A processor uses a state transition rule to change the state of its process. A state is a collection of many state components, which include the program being executed (prog), the program counter (pc), the name-value bindings for the process (prog-id), a set of programs to be evaluated as responses to interrupts (hp), and a dump which is used to save the most important components of the state when an interrupt occurs. Thomas's work is an improvement upon previous state-oriented models of control since he includes enough structure in the state to describe adequately both the internal aspects of processes and the interface operations between interacting processes. The multi-processor orientation of the model conveys the concept of concurrently evolving processes in a fashion superior to models in which the parallelism is simulated using a single processor.

Chapter 2

INFORMAL DESCRIPTION OF MPEL1

In this chapter, we present an informal discussion of the control primitives and framework which constitute the multi-path control facility of MPEL1. We assume that the reader is familiar with EL1, as described in [Weg70] or [Weg72]. If not, the reader is encouraged to read Appendix 1, a brief introduction to EL1, at this point.

The control primitives appear in the language at the syntactic and semantic level of procedure calls. Formally, they are defined as objects of mode CSUBR (control-subroutine.) For each primitive, we give a pseudo-procedure heading which specifies the name, mode and bind-class of each argument and the mode of the value returned by the CSUBR. We then give an English description of its semantics. Here, we are primarily concerned with providing motivation and understanding, without attempting to be formally precise or complete. For each primitive, a precise specification of its semantics is given in the formal definition of Chapter 4.

Section 1 motivates the concept of paths of control. Section 2 discusses paths and their associated operations in detail. In sections 3 and 4 the framework provided by the control interpreter and its role in path synchronization and

scheduling is discussed. External interrupts are introduced in section 5. Section 6 is an index of the terms used in the chapter.

As the control primitives interact rather heavily with one another in the synthesis of multi-path control structures, it is difficult to exhibit complete illustrative examples until all the primitives have been presented. Thus, we postpone the latter until Chapter 3. We also defer justification of the multi-path facility and comparison with other proposals until Chapter 5.

1. PROCESSORS

Before we turn to the informal description of MPEL1, we must first discuss the concept of 'processor.' In particular, we will discuss the distinctions between program, process and processor; the multiplexing of processors; and the relationship between processor assignment and the synthesis of new control behavior.

1.1 Interpreter as Processor

In this section, we will first consider the three components of a computation as performed by a sequential computing device, namely, program, process and processor. We will then discuss the concept that an interpreter for a programming language may be considered an abstract processor for programs written in the language.

A program is the definition of a computation, i.e. it specifies a sequence of actions which may be performed to obtain a desired result. A process is the performance of the computation specified by a program. Information is usually associated with a process to specify the set of accessible memory locations and to indicate the action which is currently being performed. A processor is an agent who performs the actions which constitute the process as specified by the program. In particular, the processor updates the information associated with the process. To

illustrate, consider a program written in the machine code of a digital computer. The computation to be performed is specified by a sequence of instructions. The execution of the program constitutes a process in which information is maintained as to which instruction is currently being executed (program-counter) and the range of memory which may be addressed by the process. The processor is the central-processing-unit (CPU) of the computer. The CPU performs the actions specified by each instruction as stored sequentially in memory. After each instruction, the program-counter for the process specifies the next instruction to be executed by the processor. Hence, instructions which modify the program-counter can cause a change in the sequence of instructions executed.

In section 1.2.2, we discussed various interpreter based models which have been used to specify the semantics of programming languages. These interpreters may be viewed as processors for programs written in the language being modeled. To illustrate, the interpreter performs the actions specified by the program in the semantic space provided by the model. The evaluation of the program by the interpreter constitutes a process in which the interpreter must maintain records which specify the variables (memory locations) accessible to the program. In addition, upon completion of some action for the program, the interpreter must be able to know which action is to be performed next.

Let us consider the way in which the records described above are maintained in the interpreter model for EL1 described in [Weg70]. The interpreter uses a ROW (name-stack) to record the names of variables which are being used by the program. Each entry in the ROW contains the name of the variable and a pointer to its value, which may be on a STACK or in the heap. The information as to which action is to be performed next, however, is implicitly recorded in the control structure of the interpreter itself. For example, to evaluate the statements of a block, the interpreter uses a FOR loop which executes each statement in turn. The use of the environment of the interpreter to store information about the process being evaluated presents no difficulty if the interpreter is to be used to evaluate only one process. However, if we wish to consider an interpreter as a processor, and we desire that this interpreter be able to switch its attention from process to process, then the interpreter cannot implicitly record the control flow of any one process in its own control structure.

We postpone further discussion of the above constraint upon interpreter based models until chapter 5. In the sections which follow, we will use the terms evaluator and processor interchangeably to describe an EL1 interpreter which explicitly records the control structure of the program it is evaluating.

1.2 Multiplexing of Interpreters

In the last section, we discussed how an interpreter may be considered to be a processor independent of the process it is evaluating. Here, we relate this concept to the evaluation of multiple asynchronous processes.

We will assume that there exist some finite number of evaluators which are available for the simultaneous evaluation of sequential processes. However, we will not put a bound on the number of processes which may be considered to be evaluating concurrently, i.e. there exists no limit on the number of processes which may be logically evaluated in parallel, even though only some subset of the processes are actually being evaluated at the same time. Hence, the evaluators must be multiplexed across the processes; each one of the concurrent processes must be run on a processor at some time.

The obvious implication of evaluator multiplexing is that an evaluator must be able to switch its attention from process to process, i.e. an evaluator must be able to stop evaluating one process and start evaluating another. Hence, an evaluator must not retain information implicitly about the process it is evaluating.

To insure that each of the concurrent processes will be evaluated at some time, there must exist a mechanism which

will force an evaluator to switch its attention from one process to another. To achieve this, we will assume that each processor has some number of external interrupts associated with it. An external interrupt may be described as a signal sent from some external processor to an evaluator to indicate that some event has occurred. For example, a timer interrupt may be described as a signal from a processor which is dedicated to marking elapsed time. It is important to note that interrupts are associated with processors, not processes. An external interrupt signals an evaluator independent of the process which is being evaluated. Hence, if some process is interested in the fact that some external interrupt has occurred, then there must exist some mechanism which allows this fact to be dispatched to the interested process.

We could, of course, avoid the problems involved with the multiplexing of evaluators by simply assuming that each process has its own evaluator. Whenever a new concurrent process is created, a new evaluator can be created to effect its evaluation. The dynamic creation of processors, however, implies that either an additional processor is added or that the existing ones are actually realized by the multiplexing of some fixed number of processors at a lower level. The former is difficult to achieve since it requires the dynamic addition of new hardware. We reject the latter on two counts. First, it only serves to suppress the issues

of multiplexing. Second, we hope that our evaluators directly model a set of physical processors capable of simultaneous processing. If the evaluators are themselves concurrent processes which are multiplexed over some smaller set, then the model is incorrect. N evaluators will not represent N processors capable of simultaneous activity.

It is our thesis that the control relationships between processes can best be explicated in a language in which the user can obtain a handle on the assignment of processors to processes. This requires that the multiplexing of evaluators be made explicit in the language. In MPEL1, the multiplexing is achieved through the use of a distinguished process which is discussed in the next section and is described in detail in section 2.3.

1.3 Paths of Control

Concurrent execution is not the only control relationship which may obtain among processes. For example, a set of processes may exhibit a coroutine relationship, which requires that only one process from the set be evaluated at any given time. A coroutine process will only be evaluated when control is explicitly passed to it from the active process. Processes may also exhibit a subroutine relationship. In this case, one process creates a second one and passes control to it while indicating that the

calling process should cease evaluation. When the called process has completed, the calling process resumes execution.

It is important to note that the control relationships described above are not properties of the processes involved. Whether two processes act as coroutines, asynchronous processes or subroutines depends entirely upon the control organization which they have mutually decided upon. The control relationships may be intermixed: two processes may first act as coroutines and then later as asynchronous tasks. Hence, we will drop the semantically loaded word 'process' in favor of the term 'path.' A path of control corresponds to the dynamic evaluation of a sequential EL1 program. Paths are not inherently asynchronous processes or coroutines, although paths may exhibit these relationships, among others.

Explicit control over processor multiplexing plays a vital role in the creation of control relationships among paths. For example, two paths may be made into asynchronous tasks by including them in the set over which the evaluators are being multiplexed. A path may be forced to cease evaluation by removing it from the set (assuming that it is not currently being evaluated.) A coroutine relationship may be established by insuring that only one of the coroutines is in the set at any time. The point is that a control

relationship between paths is essentially a specification of the way in which processors are to be assigned to the paths involved. If a language does not allow for explicit assignment, then it must be achieved by some circumlocution.

MPEL1 provides a framework in which the assignment of processors to paths can be explicitly controlled by the programmer. This is achieved through the use of the control interpreter path (CI). The CI gives interpretation to the control relationship which is to obtain among a set of paths. It contains data structures that indicate which paths are currently being evaluated and which paths are eligible for concurrent evaluation. Other paths may access these data structures, and thus affect the assignment of processors to paths. Consequently, the control interpreter path, in conjunction with its associated control primitives, provides the handle on processor multiplexing which is necessary for the synthesis of arbitrary behavior patterns among paths.

2. PATHS

In the last section, we described a path of control as the dynamic evaluation of a sequential EL1 program. Here, we will give a more precise definition of the term 'path.' In addition, we will introduce the control primitives which are applicable to paths.

Whenever a language admits multiple paths of control, a number of related issues, such as data sharing and synchronization, must be discussed. Hence, a number of the subsections below are devoted to these path related issues.

2.1 Informal Description of a Path

The evaluation of an EL1 program is a sequential process in which the flow of control can be modified by procedure calls, compound forms (blocks), conditionals, gotos, and FOR loops. Certain data structures must be maintained by the evaluator to record the control history of the evaluation. These records include information as to which procedures have been entered, which right hand sides of conditionals have been followed, etc. The records are necessary so that the evaluator may know how to continue evaluation of the program upon completion of a control modifying operation. An MPEL1 path is the union of the data structures required by the evaluator to effect the evaluation of an EL1 program. Note that one of the data

structures required is a representation of the program whose evaluation constitutes the path of control. In particular, associated with each path is an environment and an activation record.

A path's environment consists of two related parts: the identifier environment and the intra-path control.

The identifier environment contains all the name-value pairs accessible to a program at a given point in its evaluation. Name-value pairings are created by procedure application or by explicit declaration. For procedure application, the names correspond to the formal parameters of the procedure and the values are the values obtained by evaluating the corresponding actual parameters of the particular call. For explicit declaration, the names are the identifiers listed in the declaration and the values are the values obtained by repeated evaluation of the initialization form, or default values of the appropriate mode if no initialization form is specified, c.f. Appendix 1. Name-value pairs are removed from the identifier environment upon exit from the corresponding procedure call in the case of formal parameters and upon exit of the block in which declared in the case of explicit declarations.

The intra-path control contains the records associated with the control history of the path. The records constitute a partial history of control within the path

which includes all procedure calls which have not yet been completed and all blocks which have not yet been exited. The intra-path control must be related to the identifier environment so that the evaluator may update the latter when necessary, e.g. on block exit. Note, however, that the records kept do not constitute a complete history; there are no records of completed procedure calls or blocks which have been exited.

The design of EL1 allows an evaluator for the language to maintain its data structures using a stack discipline. The multi-path facility has been designed to preserve the stack discipline for sequential programs. Thus, both components of an MPEL1 path's environment are managed as stacks.

The activation record (ACTRC) of a path is defined as an EL1 STRUCT. It serves as a system 'handle' on the path. The components of an ACTRC contain vital information about the path. In particular, the location and size of a path's (stack) environment is stored in its ACTRC. The components of an ACTRC may be grouped into two classes:

- (1) Those components which may be modified only by the control primitives, and hence may only be read by a program.
- (2) Those components which may be read and written by the program to effect communication with the

control primitives.

The various components of an ACTRC will be introduced, as needed, in the sections that follow. The complete definition of the mode ACTRC appears in section 4.2.1. The class to which a given component belongs will be obvious from the discussion in which it is introduced.

A given control extension may require new fields to be added to activation records. For example, a scheduling algorithm which associates priorities with paths may require an additional integer component in which the path's priority is to be stored. We will refer to such components as extended components as opposed to the basic components of the original definition of ACTRC. The implementation of extended components, without any loss of notational convenience, can be achieved through the use of the extended mode definition facility of EL1, c.f. 5.1.2.

All activation records are allocated in the heap and thus may be referenced by pointers. The mode ARPTR is defined as a PTR(ACTRC) for convenience in discussing paths. ARPTRs have a practical value as well: two paths are identical if and only if their ARPTRs are equal, a simple pointer comparison.

2.2 Path Creation and Deletion

A path is created by calling upon the control primitive GET\PATH.

```
GET\PATH<-CSUBR(SIZE:INT;ARPTR)
```

The integer argument specifies the amount of core (in K) to be initially allocated for the path's environment. GET\PATH allocates the environment and activation record for the path and returns a pointer to the path's ACTRC. The boolean components STKEFLG (stack-environment-flag) and ELGFLG (eligibility-flag) of the path's ACTRC are set to TRUE to indicate that the path possesses an environment and that the path may be evaluated by a processor, respectively. If ELGFLG is TRUE we say that the path is eligible for evaluation. In addition, the path is enabled for certain path-level interrupts and certain fields of the ACTRC are initialized to meaningful default values. These settings will be described at appropriate points in the sections that follow.

When a path is no longer needed it may be explicitly deleted by calling upon the control primitive DELETE\PATH.

```
DELETE\PATH<-CSUBR(PATH:ARPTR;NONE)
```

DELETE\PATH reclaims the path's environment, if possible,

and sets the boolean component `ELGFLG` to `FALSE`.^{*} Once deleted, a path is no longer eligible for evaluation and therefore an error occurs if an attempt is made to pass control to it. Note that the `ACTRC` is retained as long as it is referenced by an eligible path. A path may not perform self-deletion, i.e. a call to `DELETE\PATH` with itself as argument, as this would require control to be returned to an ineligible path. Self-deletion requires a call upon the control interpreter path, c.f. 2.3.2.

2.3 Path Initialization

`GET\PATH` creates an environment in which a computation may be performed but does not indicate what is to be computed. In order for paths to be of any use there must exist a mechanism for specifying the program which is to be evaluated in the path's environment. The primitive control functions `PAP` (path-apply) and `PAPQ` (path-apply-quoted) are used to initialize the computation that the path is to perform. The relation between these two functions is the same as the relation between `SET` and `SETQ` is LISP [We67];

*

It is not the case that `STKEFLG` is `TRUE` if and only if `ELGFLG` is `TRUE`, since the environment of a path which has been deleted may have to be preserved if there exist paths which are dependent upon it, c.f. 2.2.8. It is true, however, that if `ELGFLG` is `TRUE` then `STKEFLG` must be `TRUE`; a path which is eligible for evaluation always has an environment associated with it.

the former evaluates its first argument while the latter
 does not.*

```
PAP<-CSUER(F:FORM,P:ARPTR;ARPTR)
```

```
PAPQ<-CSUBR(F:FORM UNEVAL,P:ARPTR;ARPTR)
```

Since the only distinction between PAP and PAPQ is in the bind class of their first argument, the following discussion will only reference PAP, the interpretation for PAPQ being derivative.

Let Q denote the path which has called PAP. The first argument (F) to PAP specifies a procedure call to be applied in the environment of the path which is the second argument (P) to PAP.** If P is identical to Q, then the procedure call is evaluated in the environment of Q. If P is not eligible for evaluation (P.ELGFLG=FALSE) then an error is generated in path Q.

Let $F = G(A1, A2, \dots, AN)$. The interpretation of the procedure application is as follows:

*

If we define a procedure QUOTE which returns its single argument unevaluated, then

```
PAP(QUOTE(FOO(X,Y),P))=PAPQ(FOO(X,Y),P).
```

QUOTE can be defined trivially in EL1 as follows:

```
QUOTE <- EXPR(X:FORM UNEVAL; FORM) X;
```

**

If the form F is not a procedure call, then the environment of P is modified so that if control passes to it, then the form will be evaluated in P's environment.

- (1) G is evaluated in the environment of Q to produce a procedure body G'.
- (2) The formal parameters of G' are bound to the actuals A1, ..., AN as if the procedure was to be applied in Q.
- (3) The bindings of the formals of G' are copied into the environment of P, except for bindings to objects which lie in the heap, in which case no copy is made.
- (4) The environment of P is modified so that if control passes to P, then the body of G' will be evaluated.
- (5) PAP returns a pointer to path P as result.

Note that PAP only modifies the environment of a path; no transfer of control is performed.

The following example illustrates the effect of PAP upon the arguments to the PAP'ed procedure call.

```

BEGIN
DECL A:INT BYVAL 3;
DECL B:PTR(INT) BYVAL ALLOC(INT LIKE 2);
DECL C:INT BYREF VAL(ALLOC(INT LIKE 1));
DECL P1:ARPTR BYVAL GET\PATH(1);
DECL FOO:ROUTINE;
FOO<-EXPR(V:INT BYVAL,W:INT BYREF,X:PTR(INT) BYREF,
          Y:INT BYREF,Z:INT BYREF;INT)
  BEGIN
    V+W+VAL(X)+Y+Z
  END;
PAP(FOO(A,A,B,C,A+A),P1)
END;

```

Figure 2-1 displays the state of the paths just before

the PAP is evaluated. Note the difference between the values of B and C. B is of mode PTR(INT). The value of B (a pointer to an integer in the heap) is in the environment of path Q. C is of mode INT. The value of C, however, is not in the environment of path Q. It is in the heap. Figure 2-2 displays the state of the paths just after the PAP has been evaluated. All of FOO's arguments, except for C, have been copied into the environment of P1. Note that although the value of B has been copied into path P1, the two paths may both reference the integer pointed to by B.

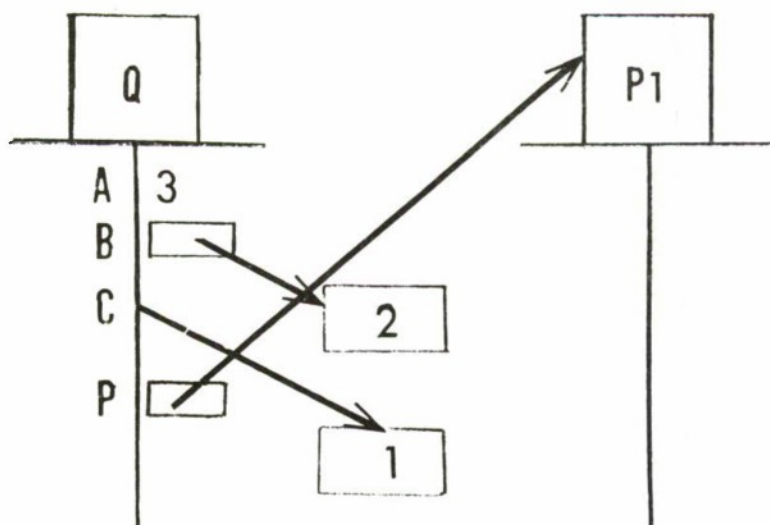


Figure 2-1 Paths Q and P1 Before PAP

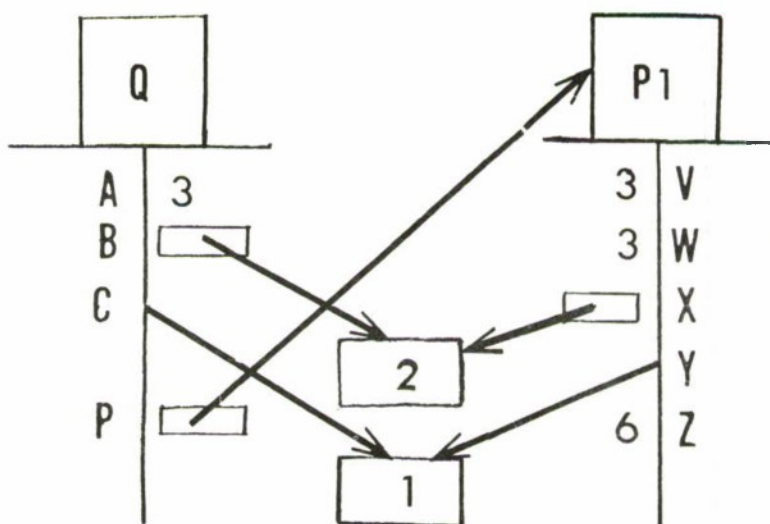


Figure 2-2 Paths Q and P1 After PAP

The treatment of arguments to PAPed procedures insures that the environment of the path PAPed into will not contain references to the environment of the path performing the PAP. If a path could reference the environment of another, then some mechanism would have to be employed to insure that the environment remained intact as long as the path retained a reference to it. Thus, it would be necessary to impose control restraints upon the paths involved. A control regime in which paths may obtain such references can be realized using the control primitives described in section 2.2.8.

The use of PAP is not restricted to initialization only; PAP may be used to apply a procedure in the environment of a path which has already started a computation. The programmer must provide the synchronization necessary to insure that the path PAPed into is not being evaluated at the time that the PAP is performed, c.f. 1.1.3. If many procedure calls are PAPed into the environment of a path, then they are executed in the reverse of the order in which they were PAPed. When the evaluation of the body of a PAPed procedure is completed, then evaluation of the path continues from the point it was at when the PAP originally occurred. A user defined path termination function is called upon exit from the outermost PAPed procedure, c.f. 2.2.6.

2.4 Path Evaluation

The two preceeding sections have specified the primitives for path creation (GET\PATH) and initialization (PAP.) No mechanism has yet been introduced which allows control to be passed to a path. An initialized path is eligible for evaluation but is certainly not being evaluated. Thus, we must specify the way in which evaluators are assigned to paths. Since it is possible to create an arbitrary number of paths which are to be evaluated concurrently by a bounded number of evaluators, the assignment must surely involve some notion of scheduling. Path scheduling is described in detail in section 2.3.5. This section introduces the terminology to be used in discussing multiple paths of control.

A scheduler is a mechanism for multiplexing the evaluation of paths by a fixed number of evaluators. A scheduler uses a scheduling algorithm to choose a path to be evaluated from a set of paths which are available for evaluation. A path is active if it is currently being evaluated. A path is inactive if it is not active but is contained in the set of paths from which the scheduler chooses. A path is running if it is either active or inactive. A path is stopped if it is not running. A path is reachable if it is running or if its ARPTR is accessible from the environment or activation record of a reachable

path. A path is lost if it is not reachable.

A created and initialized path is initially stopped. It may become a running path by explicitly including it in the set of inactive paths, c.f. 2.3.5. A path which has been deleted is no longer eligible for evaluation and therefore the scheduler will not allow it to become active.

The inclusion of inactive paths in the set of running paths requires some explanation. If a path isn't currently being evaluated, then it certainly isn't running on a processor. The classification is justified by the fact that the scheduling of paths is essentially transparent to the paths being scheduled, hence it isn't possible for a given path to determine (without some special action) which paths are active and which are inactive. Thus, the running paths are those paths which are being evaluated concurrently, although only the active paths are being evaluated simultaneously.

It is sometimes desirable to explicitly remove a path from the set of running paths. For example, if a path is to cease further evaluation until some condition is true, then it can be temporarily removed from the running set to insure that an evaluator will not be assigned to it. When the condition becomes true, then the path can be returned to the set of inactive paths. Evaluation of the path will continue as soon as it is made active by the scheduler.

It is also desirable to have the ability to indicate that a path should not become active without explicitly removing it from the set of inactive paths. For example, if a path Q wishes to PAP into another path P, it could check the set of running paths and remove P from the set to insure that it will not become active.^{*} But suppose that P is not in the running set. Q cannot safely perform the PAP because it is possible that some other path will asynchronously include P in the inactive set. Hence, P may become active while the PAP is being performed. Activation records contain the boolean component DORMANT which is used to indicate that a path should not be allowed to become active. The scheduler will not assign an evaluator to a path with DORMANT=TRUE. If DORMANT is set to TRUE while a path is active, then once the path becomes inactive it will not become active again until DORMANT is set to FALSE.

2.5 Data Sharing

It may be necessary for a set of paths to access common data structures in order to collectively perform a given computation. In general, paths may not share data by referencing the environments of other paths, unless the paths involved are willing to constrain their control relationships to an organization in which such sharing is

*

If P is active, this may take some time, c.f. 2.5.5.

feasible, c.f. 2.2.8. Thus, most data structures shared between paths will lie in the heap. In this section, we will discuss the various means by which paths may share data.

All paths are embedded in a global, or top-level, environment. The global environment consists of name-value pairs in which all values lie in the heap. If a path references a variable which is not currently defined in its environment then the reference is taken to be to the value of that variable in the global environment.^{*} Since all paths have the same global environment, sharing can be achieved by referencing the same top-level variables.

Sharing may also be achieved by using the control primitive PAP. Since the arguments to a PAPed procedure call are evaluated in the environment of one path and the procedure call in another, PAP provides a mechanism which allows sharing relations to be established at the time a path is initialized. There are two cases of interest. First, a path may pass a pointer as argument to a PAPed procedure call. Although the pointer is copied into the environment of the path PAPed into and thus is itself not shared, the object to which it points is accessible from both paths. In the second case, a path may pass an argument

*

The method by which global variables are initialized is discussed in Appendix 1.

which is bound directly to an object in the heap. In this case, if the bind class is BYREF, then PAP will pass the argument directly to the PAPed path without making a copy. Hence, both paths reference the same object.

A path may obtain the value of a variable defined in the environment of another path via the control primitive PFETCH (path-fetch.)

PFETCH<-CSUBR(NAME:SYMBOL,P:ARPTR;ANY)

PFETCH searches the environment of path P for the most recent occurrence of the variable NAME and returns either the value of the variable (if the value lies in the heap) or a copy of the value of the variable (if the value is in the environment of P.)

A path may change the value of a variable defined in the environment of another via the control primitive PSTORE (path-store.)

PSTORE<-CSUBR(NAME:SYMBOL,P:ARTPR,VAL:ANY;NONE)

PSTORE searches the environment of path P for the most recent occurrence of the variable NAME and replaces its current value with VAL. (An error occurs if VAL cannot be converted to the mode of the value of NAME.)

Both PFETCH and PSTORE require that the path P be not active. If P is active, then an error occurs in the environment of the path performing the PFETCH (PSTORE). If P is inactive but becomes active while the search is being

performed, then an error occurs in the environment of the scheduler.* A more useful error is generated if the variable NAME does not exist in the environment of P. In this case, the programmer may supply (via MPEL1 error handling) another path to be searched through. Thus, it is possible to construct arbitrary searches through a given set of paths.

Although PFETCH and PSTORE will usually be used to update or copy pointers to objects in the heap, they also provide a mechanism, although inefficient, whereby a path can share an object in the environment of another. Assuming the appropriate synchronization is available, a path P can stop another path Q, obtain a copy of the value of a variable contained in Q's environment, modify the copy, use PSTORE to replace the original with the updated version and then allow Q to continue evaluation. The two paths are effectively sharing the object since all modifications made by both paths will be reflected in the data.

2.6 Path Termination

A path is terminated if it is no longer eligible for evaluation. The control primitive DELETE\PATH makes a path ineligible for evaluation since it sets ELGFLG to FALSE and usually deletes the path's environment. DELETE\PATH is of

* P is being modified, thus it cannot become active, c.f. 1.1.3.

limited usefulness by itself, however, because the above is all that it does. The fact that a path has terminated may be of interest to other paths. DELETE\PATH provides no mechanism for broadcasting the path's demise. A path may desire, upon termination, to return a value to some set of paths which are waiting for its value. DELETE\PATH does not provide for a value to be associated with a path.

The capabilities described above can be achieved, however, by using DELETE\PATH in conjunction with other control facilities of MPPL1. A procedure can be written which will cause a path to wait until a given path terminates and another can be written which will cause explicit termination of a path along with notification to all waiting paths, c.f. 3.3.

There is one point still remaining. When a path exits the outermost procedure call in its environment, it is probably trying to indicate that it would like to be terminated. In addition, it might be useful to specify the value returned by the outermost procedure call as the value of the path itself. Although it would seem desirable to allow a path to terminate itself implicitly in this way, the termination procedures described above must be called explicitly. The solution is straightforward. We include the component TERMINATION\FORM as one of the fields of an activation record. When a path exits its outermost

procedure call, the value returned by the procedure is bound to the identifier "LAST\VALUE" and then the TERMINATION\FORM is evaluated. The form can save the last value, notify any waiting paths and call DELETE\PATH to actually terminate the path. The TERMINATION\FORM is initially set by GET\PATH to be a procedure call which will cause DELETE\PATH to be called. Since DELETE\PATH usually deletes the path's environment, the last value should be copied into the heap, if it is not there already. Otherwise, the value will be lost when the path's environment is reclaimed.

2.7 Path Synchronization

If paths are allowed to evaluate concurrently, then they must be provided with a mechanism which permits them to synchronize their activities. For example, if a path P desires to cease evaluation until another path Q terminates, then it might first test the value of Q.ELGFLG and then add itself to a queue of paths waiting for Q to terminate, viz.

```

Q.ELGFLG -> BEGIN
              Put self on queue associated with Q;
              Cease evaluation
            END

```

Let us assume that when Q terminates it indicates that all paths on its queue may become active. If Q terminates after P tests Q.ELGFLG but before the block above is evaluated, P will never be awakened! P and Q must be able to synchronize their actions, i.e. P must insure that Q does not terminate

while it is in the process of performing the wait and, conversely, Q must insure that P does not try to wait while it is in the process of termination. The way in which paths may effect synchronization is discussed in detail in section 2.3.2. In this section, we will discuss synchronization with respect to the control facilities described earlier.

Let us first consider synchronization in relation to the path scheduler. If the scheduler has access to more than one evaluator, then it is obvious that a mechanism for path synchronization is necessary. However, if the scheduler is multiplexing paths using only one evaluator then a synchronization facility is still necessary since the scheduling of paths is transparent to the running paths. In particular, in between testing Q.ELGFIG and queuing itself, P may become inactive and Q may become active. If Q terminates, then the situation is as disastrous as if P and Q had been active simultaneously.

The control primitives PAP, PFETCH, and PSTORE contain no built in synchronization. For example, if two paths try to concurrently PAP into the same path, then the system will not insure that first one PAP will occur and then the other. In fact, an error will be generated to indicate the lack of synchronization, c.f. 1.1.3. Thus, unless the organization of one's paths is such that it is impossible to perform concurrent PAPs into a path, one must provide a layer of

synchronization around the control primitive, c.f. 3.3. The general rules with respect to the use of PAP, PFETCH and PSTORE are as follows.

- (1) The affected path should be not active and should not be allowed to become active until the control primitive has completed its action.
- (2) Only one of the control primitives may be applied to a path at a given time.

The control primitive TSET (test-and-set) may be used for path synchronization as an alternative to the mechanism described in section 2.3.2. TSET is defined as follows.

```
TSET<-CSUBR(X:INT;BOOL)
  BEGIN
    If X is 0, then set X to 1 and return TRUE.
    Otherwise, return FALSE.
  END;
```

TSET is an indivisible operation with respect to a set of active paths; if two paths simultaneously TSET the same integer, whose current value is 0, then TSET will return TRUE to one path and FALSE to the other. Variations on TSET [La68] [IBM68] have been described in the literature. TSET is included as an MPEL1 control primitive because it provides the most basic mechanism for inter-path synchronization. In order for it to be used for synchronization, however, TSET requires that a path go into

a loop continuously calling it until TRUE is returned. ^{*} This phenomenon is known as the busy wait and is obviously quite wasteful. The energy of the evaluator would be better spent upon a path which could do some useful work. The facility described in section 2.3.2 allows for path synchronization at a much lower cost since it provides for a nonbusy wait.

2.8 Path Dependency

All MP_{EL}1 paths discussed so far may be considered to be independent in the sense that no path can directly reference the stack environment of another. This phenomenon is a result of the fact that the control primitive PAP copies all arguments which would ordinarily be passed BYREF and of the fact that the control primitive PFETCH returns a copy of the value of its argument. It is sometimes useful, however, to organize a set of paths in a tree structure in which a path P may directly reference the stack environment of some path Q which is higher in the tree. In this situation we may say that P is dependent upon Q in the sense that P requires Q's environment in order to evaluate properly. In this section, we will explore the concept of path dependency and introduce the control primitives necessary to establish this path organization.

*

To unset the integer one uses the control primitive CLEAR, which sets the integer to 0. Although CLEAR does not have to be defined as primitive, it is included for symmetry.

A path is initially independent. One path can cause another path to become directly dependent upon it by calling upon the control primitive MDEP (make-direct-dependent.)

MDEP<-CSUBR(P:ARPTR;ARPTR)

When a path P becomes a direct dependent of another path Q, it may then reference the entire identifier environment of Q up to and including all variables defined in Q at the point at which MDEP was called. MDEP returns a pointer to the path which has become the direct dependent. MDEP generates an error if $P=Q$, if P is already the direct dependent of some path other than Q, or if a circular dependency would be created.

The following definitions will simplify the discussion of path dependency. A path P is dependent upon a path Q if and only if either P is a direct dependent (dd) of Q or if there exists paths P_1, \dots, P_n such that

$P \text{ dd } P_1 \text{ dd } P_2 \dots P_n \text{ dd } Q$

If P directly depends upon Q, then Q directly supports P. If P depends upon Q, then Q supports P. The sub-environment of a path Q which may be referenced by a directly dependent path P is the directly accessible environment of Q with respect to P. The accessible environment of a path P is the union of the directly accessible environment of P with all environments directly accessible to the supporters of P.

We may now restate the conditions under which a path Q

may make a path P its direct dependent:

- (1) P is not dependent on any path (except perhaps Q.)
- (2) Q does not depend upon P.
- (3) P and Q are not the same path, i.e. $P \neq Q$.

A path may obtain a reference to a variable in its accessible environment by calling upon the control primitive DEPENV.

DEPENV ← CSUBR(X:SYMBOL;ANY)

DEPENV searches for X in the environment of the path in which it is called and if it is not found then it searches for it in the accessible environment of the path. If no X is found in the accessible environment, then the global value of X is returned as the result of DEPENV. The restriction on MDEP^{*} that P is not directly dependent upon any path except Q allows DEPENV to be a single valued procedure since a path can be directly dependent upon only one path.

The control primitive DPAP is defined as follows.

DPAP ← CSUBR(F:FORM,P:ARPTR;ARPTR)

The effect of DPAP is identical to that of PAP except for the following modification: if P is dependent upon the path performing the DPAP, then all arguments passed BYREF to the DPAPed procedure which reference the accessible environment of P are passed directly to it, i.e. no copy is made.

*

The effect of multiple MDEPs is to extend the environment of Q which is directly accessible to P

The concept of directly accessible environment requires closer examination. In particular, three points are of interest: a precise description of the directly accessible environment, the observation that the direct dependents of a path may have different directly accessible environments, and the restrictions imposed upon the intra-path control of a supporting path.

The directly accessible environment corresponds precisely to the identifier environment of the path when MDEP is called, i.e. it is composed of all the variables which could be referenced by the path at the point MDEP is called. For example, consider the following block:

```

BEGIN
  DECL X:INT BYVAL 4;
  BEGIN
    DECL Q:ARPTR BYVAL MDEP(A);
    DECL R:INT BYVAL 5;
    DECL S:ARPTR BYVAL MDEP(B);
    DECL T:INT BYVAL 6;
    FOO(R,X,MDEP(C));
    EXPR(M:INT,N:INT;INT)(FUM(M+N,MDEP(D)))(R,X);
    MDEP(B);
    .
    .
  END;
END;

```

The directly accessible environments of A,B,C and D are:

- A: X, ... Q is not included since it is not in the identifier environment when MDEP is called.
- B: R,Q,X, ... Same as for A , with respect to S.
- C: T,S,R,Q,X, ... All declarations are included, but the formals of the procedure FUM are not, since they are not included in the environment until the procedure is entered.
- D: N,M,T,S,R,Q,X... The formals of the literal procedure are included, but the formals of FUM are not.
- B: T,S,R,Q,X, ... The second MDEP(B) allows B to reference T and S.

From the example above it should be obvious that the direct dependents of a given path may have different directly accessible environments. Note that the effect of the second MDEP(B) is to extend the portion of the environment of the path which is accessible to it.

The intra-path control of a supporting path must be constrained so that no portion of its identifier environment is deleted until all dependents, who can access that portion, are terminated. In the example above, the path may not exit the inner block until B,C, and D have terminated. It may not exit the outer block until A has terminated. A supporting path has essentially three options: terminate all dependents who can reference the sub-environment which is about to be deleted; wait until all dependents have terminated before deleting the sub-environment; and terminate itself, in which case the environment of the path

will be retained until all dependents have terminated. The last case invokes the situation described in section 2.2.2, where ELGFLG becomes FALSE but STKEFIG remains TRUE. If a path attempts to delete a portion of its environment which is accessible to a non-terminated dependent, then an error is generated with the accessible environment still intact. The error may be handled by the programmer. To facilitate this, a simple recursive procedure can be written to determine which of the dependents must be terminated, c.f. 5.1.3.

2.9 Intra-Path Control Primitives

There are five primitives which are primarily concerned with intra-path control: GOTO, RETFROM, MYPATH, COPY and EVAL.

In order to understand the use of GOTO we must first discuss the treatment of labels in EL1. Each statement in an EL1 block may have one or more labels associated with it, e.g.

L1: L2: FOO(A,B) => TRUE

All labels are implicitly DECLared to be variables of mode LABEL as the last declarations of the block in which they appear. It is also possible to explicitly declare a variable to be of mode LABEL and to pass a label valued variable as a parameter to a procedure. It is not possible,

however, to assign to a label variable or to return an object of mode LABEL as the result of a procedure.

GOTO is defined as follows:

```
GOTO<-CSUBR(L:LABEL;NONE)
```

GOTO modifies the environment of the path so that evaluation will continue with the statement specified by the label L in the most recent incarnation of the block in which L was declared. Note that L must specify a block which has been entered by the path performing the GOTO, i.e. it is not possible to pass control between paths by calling GOTO in one path with a label which references the environment of another. Transfer of control between paths is achieved through the use of the control interpreter, c.f. 2.3.1.

The control primitive RETFROM

```
RETFROM<-CSUBR(FNAME:SYMBOL,VAL:ANY;NONE)
```

is used to return control from the most recent explicit call on the procedure FNAME with VAL as result. If the environment of the path does not contain an explicit call on

*

An explicit call on a procedure is one in which the form which is to evaluate to a procedure body is of mode SYMBOL, e.g.

```
FOO(A,B,C)
```

is an explicit call on FOO, whereas

```
BEGIN TRUE => FOO END (A,B,C)
```

is not.

FNAME, then an error occurs.

Since GOTO and RETFROM can cause portions of the identifier environment of a path to be deleted, they must be used with caution in a supporting path. For example, a local GOTO, i.e. within a block, presents no problem, but a non-local GOTO will induce a runtime error if there exist non-terminated paths which are dependent upon the environment deleted, c.f. 2.2.8.

The control primitive MYPATH

MYPATH<-CSUBR(;ARPTR)

returns a pointer to the activation record of the path in which it is called. Since MYPATH has a null argument list, it is defined as a NOFIX operator [Weg72], i.e. it may be called without being followed by an empty set of parentheses, e.g.

Y<-MYPATH.TERMINATION\FORM

as opposed to

Y<-MYPATH().TERMINATION\FORM.

The control primitive COPY

COPY<-CSUBR(P:ARPTR;ARPTR)

creates a copy of the path specified by P and returns a pointer to the activation record of the new path. If P was directly dependent upon some path Q, then the new path, say T, is also directly dependent upon Q. P and T have precisely the same accessible environments. T does not,

however, support the same paths as P. In fact, it does not support any path.^{*} Thus, although P and T have identical environments, their interpretations as paths are slightly different.

The control primitive EVAL

EVAL ← CSUBR(F:FORM;ANY)

evaluates the form F in the current path's environment. EVAL returns as result the value obtained by evaluating the form.

*

If T was to support the same paths as P, then it would be possible for a path to be directly dependent upon two paths.

3. THE CONTROL INTERPRETER

In the last section, we discussed the path related issues of scheduling and synchronization while postponing the explanation of how they are resolved in MPOL1. Here, we introduce the control apparatus necessary to resolve these issues.

There exists one distinguished path in MPOL1, the control interpreter (CI) path. The CI path is composed of an environment and an activation record, just like any path which has been created by GET\PATH. In particular, the global variable PCIR is bound to a pointer to the control interpreter's activation record. The program being evaluated in the environment of the CI may be defined in EL1. However, the distinguishing feature between the CI and its fellow paths is that it is the only path to which other paths may directly pass control and it is the only path that may directly pass control to a path other than the CI. In the following sections, we will discuss how the CI path may be used to provide a mechanism for path scheduling and synchronization.

3.1 Communication with the CI

A path may pass control to the CI by calling upon the control primitive CIA (control-interpreter-apply.)

```
CIA<-CSUER(FN:ONEOF(SYMBOL,ROUTINE),ARG:ANY;REF)
```

If FN is a symbol, then it specifies the name of a procedure to be applied in the environment of the CI. If FN is a ROUTINE, then it specifies the body of the procedure to be applied. ARG specifies the argument to the procedure. If the mode of ARG is not of class PTR, then ARG is copied into the heap and the argument to the procedure is taken to be a pointer to the copy.

The CIA call is carried out as follows:

- (1) FN and ARG are stored in the components of the path's activation record named CIA\FN and CIA\ARG respectively. Both components are of mode REF.
- (2) Control is transferred from the environment of the path to the environment of the CI. Although it is possible that the CI is currently active, we postpone discussion of this case until the next section. Hence, let us assume that the CI is not active.

The program in the CI path then performs the following actions.

- (a) If CIA\FN is a SYMBOL, then it is evaluated to produce a procedure to be applied.
- (b) The procedure obtained in (a), or CIA\FN itself, is applied to the argument specified by CIA\ARG.
- (c) Control is passed back to the path which performed

the CIA. Although it is possible to pass control to a path other than the one which performed the CIA, we defer explanation of this ability until after we have described the environment of the CI.

- (3) When control is returned to the path, the CIA primitive returns the REF specified by the CIA\RESULT component of the path's activation record. Hence, a CIA called procedure may effectively return a result in the environment of the path which performed the CIA by assignment to the CIA\RESULT component of the path's activation record.

At this point, it may be useful to discuss briefly the differences between PAP and CIA, since they both have the effect of generating a procedure application in the environment of another path. First, CIA may be used only to apply a procedure in the environment of the CI, whereas PAP may be used to apply a procedure call in any path. Secondly, CIA requires that the procedure to be applied take exactly one argument of mode class PTR, whereas the arguments to a PAPed procedure are not restricted with respect to number or mode. Lastly, CIA transfers control to the CI to apply the procedure; PAP, on the other hand, never transfers control between paths.

The limitation that a CIA called procedure take only one argument (of mode class PTR) also requires some explanation. We have limited it in this way since a call in which the procedure takes an arbitrary number of arguments (with no restrictions on the modes of the arguments) can be achieved by extension, c.f. 5.1.1. In addition, the passage of multiple arguments to a CIA called procedure requires the construction of a list of the arguments even in the case where only one argument is passed. Since CIA called procedures usually ^{*} require only one argument, it would seem counter-productive to build in a mechanism which is wasteful in the common case.

We have not yet specified how the CI returns control to the path which has performed the CIA call. The return is achieved by calling upon the control primitive CONTPATH (continue-path)

CONTPATH<-CSUBR(P:ARPTR;ARPTR)

which may only be called in the CI environment. CONTPATH inspects P's activation record to determine whether or not P may become active. It may not become active if any of the following are true:

- (1) P.ELCFLG=FALSE. (P has been deleted.)
- (2) P.DORMANT=TRUE. (P is temporarily restrained from evaluating.)

*

Examine the examples in the next chapter.

(3) P is currently being modified.*

If none of the above conditions hold, then control is transferred from the CI to the path, otherwise an error is generated in the CI. CONTPATH TSETs the MOD field of the path's ACTRC to indicate that the path is being modified.

When a path Q passes control to the CI for a CIA call, it is essentially performing a RETFROM("CONTPATH",Q) in the environment of the CI; in other words, the result returned by CONTPATH is the ARPTR of the path performing the CIA call. Note, however, that returning control from the CI to a path P does not necessarily have the effect of performing a RETFROM("CIA",P.CIA\RESULT) because it is possible that while control resided in the CI, one or more procedures have been PAPed into P's environment. Hence, execution in P will continue with the evaluation of the body of the last PAPed procedure or with a return from the CIA call if no such procedures exist.

3.2 Synchronization

In the last section, we postponed discussion of the effect of a CIA call in the case where the CI is already active. Here, we discuss the interpretation and

*

P is being modified if it is active or being PAPed into, or being PFETCHed from, etc. In general, P is being modified if P.MOD has been TSET, c.f. 1.1.3.

implications of such a call.

In section 2.2.1, we characterized a path as the union of the data structures required by an evaluator, i.e. a path is the set of records which must be maintained to effect the evaluation of a sequential EL1 program by a single evaluator. Although a path may be evaluated by different evaluators during its lifetime, it may be evaluated by only one evaluator at any instant. Thus, it is not logically admissible for two or more evaluators to be evaluating the same path simultaneously. In particular, the CI may be evaluated by only one processor at a time. Hence, if two paths attempt to pass control to an inactive CI, then one will actually achieve passage while the other will be forced to wait. When the CI becomes inactive, as a result of a call to CONTPATH, then the waiting path may pass control to it. Consequently, the CI acts as a single access resource with respect to other paths.

The role played by the CI path in the construction of synchronization operations should now be obvious. The operation of any procedure which is only called in the CI environment is indivisible with respect to calls on that procedure by other paths, i.e. if two paths both call the same procedure, then the execution of one call will be completed before the execution of the other is allowed to begin. Any actions which require indivisible operation can

simply be done in the CI environment. For example, consider the problem of path termination discussed in section 2.2.7. If P wishes to wait for Q to terminate, it CIA calls a procedure which checks to see if Q has already terminated. If it has terminated, then the procedure allows control to flow back to P, otherwise the procedure puts P on a queue associated with paths waiting for Q and indicates to the CI that P wishes to cease evaluation. When Q terminates, it CIA calls a procedure which puts all the paths waiting on Q into the set of inactive paths and then calls DELETE\PATH. Since paths are added and removed from the queue only when control is in the CI, it is impossible for a path to cause itself to be queued forever.

3.3 The Environment of the Control Interpreter

In section 2.3.1, we introduced the control primitives CIA and CONTPATH which may be used to transfer control to and from the CI path. In this section and the next we will discuss the way in which these primitives can be used in conjunction with a set of non-primitive EL1 procedures to effect path scheduling and synchronization. Note that the organization described here consists of the conventions

*

The way in which a CIA called procedure indicates this is discussed in 2.3.4.

**

Recall that the inactive paths are those paths which would be active if there existed enough processors.

imposed by the (non-primitive) program being evaluated in the CI path, c.f. 1.1.3.

We must first describe the identifier environment in which a CIA called procedure is applied. For each variable in the identifier environment, we will give its name, mode and a brief description of its use.

DECL LASTRUN:ARPTR;

When the CIA called procedure is applied, LASTRUN contains a pointer to the ACTRC of the path which performed the CIA call. Upon completion of the CIA called procedure, the CI will pass control to the path specified by LASTRUN, unless it has been set to NIL. In this case, the CI selects an inactive path and passes control to it instead of the original path.

*

DECL INACTIVEQ:ARQPTR;

INACTIVEQ is a queue of the paths which are currently inactive, i.e. those paths which would be active if there were enough evaluators. INACTIVEQ.FIRST specifies the first path on the queue; INACTIVEQ.LAST specifies the last path on the queue. A path is linked to the next path on the queue through the NEXT component of its activation record, e.g. INACTIVEQ.FIRST.NEXT is the ARPTR of the second path on the queue.

DECL NPROC:INT;

*

ARQPTR is a mode which is defined in the global environment to be a STRUCT(FIRST:ARPTR, LAST:ARPTR).

NPROC specifies the number of processors over which the paths are being multiplexed. Thus, NPROC is an upper bound on the number of paths which may be active at the same time.

```
DECL NFPROC:INT;
```

NFPROC specifies the number of processors which are free in the sense that they are not currently being used to evaluate a path, i.e. the number of processors which are idling.

```
DECL PROCNUM:INT;
```

Each processor has a unique integer N associated with it ($1 \leq N \leq \text{NPROC}$.) PROCNUM is the number of the processor which was evaluating the path which performed the CIA call. Hence, PROCNUM specifies the processor which will evaluate the CIA called procedure.

```
DECL USER\SCHEDULER:ROUTINE;
```

The USER\SCHEDULER is the procedure which is being used to select which inactive paths should become active. As the name implies, the procedure may be supplied by the user, c.f. 2.4.

```
DECL PAVECT:ROW(NPROC,STRUCT(CURPATH:ARPTR,IDLEPATH:ARPTR));
```

For $I \# \text{PROCNUM}$, PAVECT[I].CURPATH specifies the path being evaluated by the I 'th processor. Each processor has an idling path associated with it, i.e. a path which it evaluates if it has no 'real' path to evaluate. NFPROC is the number of processors which are evaluating their idling paths. If processor K is idling, then

```
PAVECK[K].CURPATH=PAVECK[K].IDLEPATH.
```


Note that

```
PAVECT[PROCNUM].CURPATH=LASTRUN
```

i.e. the processor which is currently evaluating the CI path is the processor which was evaluating the path which performed the CIA call.

```
DECL RUNSET\FLAG:BOOL;
```

The RUNSET\FLAG is initially set to FALSE. If a CIA called procedure adds paths to the set of running paths, then it should set RUNSET\FLAG to TRUE to indicate that additional paths may have to be scheduled.

```
DECL PIVECT:ROW(NPROC,LIST);
```

PIVECT is used in conjunction with processor to processor interrupts. We postpone further discussion of PIVECT until section 2.5.5.

3.4 Path Scheduling

In the last section, we described the environment in which a CIA called procedure is applied. Here, we will describe the way in which the CI uses these data structures in the performance of path scheduling.

Let us assume that a path P has executed the following statement:

```
CIA("FOO",Q)
```

Control passes to the CI as described in section 2.3.1. However, before the procedure is applied to its argument,

the CI performs the following actions:

- (1) LASTRUN is set to P.
- (2) RUNSET\FLAG is set to FALSE.
- (3) PROCNUM is set to the number of the processor which had been evaluating P.

Upon completion of the CIA called procedure, the CI performs the following actions:

- (1) If LASTRUN is NIL, then the CI calls upon the USER\SCHEDULER to obtain a path to be evaluated by the processor. If the USER\SCHEDULER returns NIL, then the CI chooses the idle path associated with the processor, i.e. PAVECT[PROCNUM].IDLEPATH, and increments NFPROC by one. In any case, the CI binds LASTRUN to the path to which the processor is to be given.
- (2) If RUNSET\FLAG is TRUE, then the CI determines if there are any free processors (NFPROC#0) and, if so, it sends an interrupt to one of them to force it to pass control to the CI to obtain a 'real' path to evaluate, c.f. 2.5.5.
- (3) The CI sets PAVECT[PROCNUM].CURPATH to LASTRUN to indicate which path the processor will be evaluating.
- (4) The CI passes control to the path to be evaluated and positions itself to accept the next CIA call by executing the following statement.

LASTRUN←-CONTFATH(LASTRUN)

The USER\SCHEDULER is initially bound to a procedure which removes the first activation record from the INACTIVEQ and returns a pointer to it as result, i.e. as the path to be evaluated. If the INACTIVEQ is empty, then the procedure returns NIL. If paths are always added onto the tail of the INACTIVEQ by CIA called procedures, then the paths are scheduled on a 'round-robin' basis.

To obtain a better understanding of the use of the CI as single access resource in relation to its use as a path scheduler, let us again turn to our path termination example. If P wishes to cease evaluation until Q terminates, then it simply sets LASTRUN to NIL to indicate to the CI that the processor should be given to another path. When Q terminates, it appends all paths waiting for its termination onto the tail of the INACTIVEQ and then sets RUNSET\FLAG to TRUE to indicate to the CI that there are additional paths to be scheduled. Both P and Q require indivisible execution coupled with the ability to modify the scheduler's queues. The CIA call provides both of these

*

Since Q will no longer be runnable, it will set LASTRUN to NIL. In this case, it is not really necessary to set RUNSET\FLAG to TRUE, since the fact that LASTRUN=NIL will cause new paths to be scheduled anyway. Note, however, that it is possible that a path may add additional paths to the INACTIVEQ, and still wish to continue evaluation. In this case, it must set RUNSET\FLAG to TRUE.

facilities since it allows a procedure to obtain indivisible execution in the environment of the path scheduler.

In section 2.2.4, we defined a scheduler as a mechanism for multiplexing the evaluation of an arbitrary number of paths by a fixed number of evaluators. The path scheduler, as described above, does not quite fit this definition. The problem is as follows: if NPROC paths are currently active and none of the paths ever perform a CIA call, then the inactive paths will never become active. Hence, the evaluators will not be multiplexed over all paths. The solution to this problem is straightforward. If, after some given length of time, a path refuses to relinquish its evaluator, then the path's evaluation is interrupted by a "TIMER" interrupt. The response to the interrupt generates a CIA call which puts the path at the end of the INACTIVEQ and sets LASTRUN to NIL. The path scheduler can then give the evaluator to another path via the mechanism described above. A more detailed description of the way in which this 'time-out' is accomplished is given in section 2.5.5.

Since the CI is an MPEL1 path, the actions performed by it to effect path scheduling can be described by a set of EL1 procedures. These procedures are listed in Appendix 3. In the next section, we will discuss how these procedures, in conjunction with the CIA control primitive, may be used to extend the path scheduler itself.

4. USER DEFINED SCHEDULING

When a processor becomes free, the CI uses a simple algorithm to assign it to an inactive path: it is given to the first path on the queue of inactive paths. It is surely not desirable that this algorithm be the only one which may ever be used to assign processors to paths. For example, a given language application might require paths to be scheduled on the basis of associated priorities. Although it is conceivable that we could circumvent the fixed algorithm by suitably adjusting the inactive queue to insure that the next path chosen by the scheduler is the one which is desired, it would be inconvenient and inefficient to do so. Hence, we desire a mechanism which will allow both user control over path scheduling and the addition of data structures to the CI environment to support the extended scheduler.

4.1 Scheduler Extension

Three different methods may be used to extend the path scheduler: rebinding of the procedure which is called to obtain the next path to be evaluated, nesting of schedulers, and complete redefinition of the CI procedures and environment. The first of the above is exceedingly simple to accomplish but provides the weakest form of extension. The second combines the first method with the ability to

call upon the CI procedures recursively in order to obtain a nesting of schedulers. The last method requires the largest amount of work, but allows the user the ability to rewrite the control interpreter completely.

In section 2.3.3, we indicated that to obtain a path to be evaluated, the CI calls upon the ROUTINE bound to the variable USER\SCHEDULER. Hence, the scheduling algorithm can be changed by simply CIA calling a procedure which binds USER\SCHEDULER to a user defined procedure. Upon completion of the CIA call, the user's scheduling algorithm will be employed by the CI.

There are two disadvantages with this method. First, if the user defined scheduler requires additional data structures, then it must resort to the use of global variables. Secondly, there is no convenient way to nest the schedulers, i.e. if the scheduling algorithm is to be redefined more than once, then each new scheduler must understand the organization of the previous one. In addition, there is no convenient way of keeping track of how many times and the order in which the scheduler has been redefined. Hence, it is difficult to revert back to a previous scheduler once a new one has been installed. Consequently, this method of scheduler extension is of limited usefulness.

The second method of scheduler extension resolves the problems associated with the first.

The actions of the control interpreter path are embodied in the definition of one EL1 procedure, C\I. In particular, C\I applies the CIA called procedure to its argument, calls upon the scheduler if necessary, and passes control out of the control interpreter via CONTPATH. Since C\I is written in EL1, its actions are easily understood and it may be called from a user program. ^{**} If C\I is called recursively in the environment of the CI, then path scheduling and the processing of CIA calls will be performed by the inner call. C\I assumes that the variables discussed in section 2.3.3 exist in the environment in which it is called and declares local variables with the same names which are bound BYREF to their counterparts in the environment, e.g.

```
DECL LASTRUN:ARPTR BYREF LASTRUN;
```

The procedure C\I may be used to achieve a nesting of schedulers by CIA calling a procedure, say INIT\SCHEDULER, which performs the following actions.

- (1) USER\SCHEDULER is declared locally to be the

*
The definition of the procedure C\I is given in Appendix 3.

**
Although it should only be called in the CI environment, because of the call on CONTPATH.

routine to be used as the new scheduler.

- (2) Other variables that are needed by the new scheduler are declared locally.
- (3) The data structures which define the inactive set of the previous scheduler are mapped into the data structures to be used by the new scheduler. As this is a complicated process, we postpone discussion of how it can be accomplished.
- (4) The procedure C\I is called recursively. C\I will bind USER\SCHEDULER to the procedure bound locally above. Scheduling will continue in an environment which includes the data structures required by the new scheduler.

It is possible to return control over path scheduling back to the previously defined scheduler by CIA calling a procedure, say TERM\SCHEDULER, which performs the following actions.

- (1) The control primitive RETFROM is used to return control to the body of the procedure which initiated the recursive call on C\I, i.e.

RETFROM("C\I",NIL)

- (2) The data structures which define the inactive set for the current scheduler are mapped into the data structures required by the old scheduler. Again, we postpone discussion of how this can be accomplished.

- (3) The procedure returns control to the previous incarnation of C\I by a normal procedure exit. Since the recursive call on C\I bound the variables used in the previous call BYREF, the values of the variables are still valid, e.g. PROCNUM correctly specifies the number of the processor which is currently evaluating the CI path.*

Figure 2-3 illustrates the flow of control in the CI with respect to nesting of schedulers. Down arrows indicate the passage of time, right arrows indicate calls to procedures, and left arrows indicate returns from procedures. INIT\SCHEDULER is a procedure which is CIA called to initialize a new scheduler as described above and TERM\SCHEDULER is a procedure which is called to return control back to the previous scheduler.

*

This will be true only if USER\SCHEDULER is the only variable declared by the initializing procedure whose name is in common with the variables of section 2.3.3.

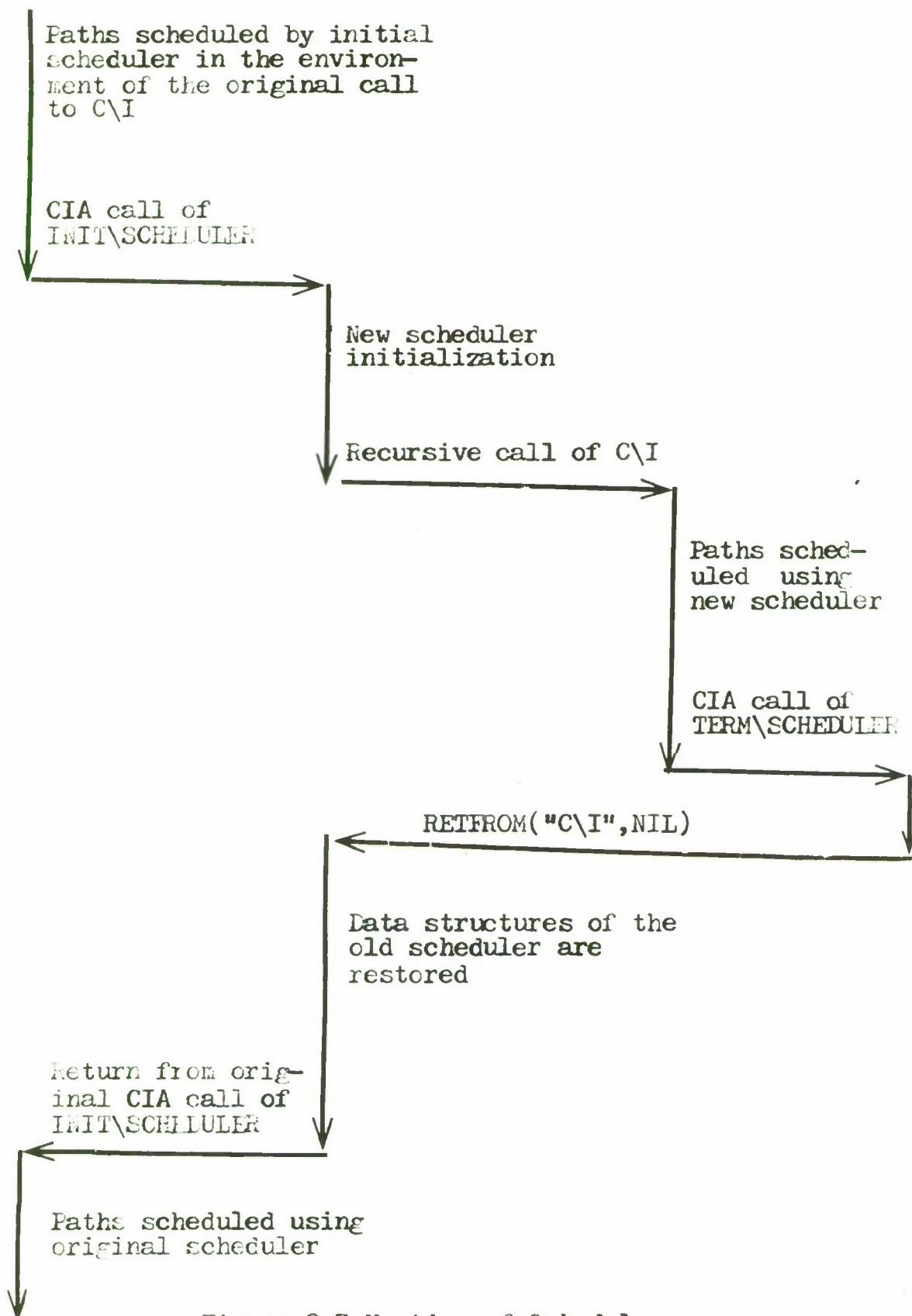


Figure 2-3 Nesting of Schedulers

The last way in which the scheduler may be extended is to rewrite the procedures which constitute the control interpreter itself. Although we believe that the organization imposed by the procedures of Appendix 3 facilitates the construction of new schedulers and synchronization operations, it is possible that another organization may be more suitable for a given class of problems. Hence, the user is free to completely restructure the CI path in terms of his own data structures, procedures and the control primitives CIA and CONTPATH.

4.2 Canonical Inactive Sets

In the last section, we postponed discussion of the way in which the data structures of one scheduler are mapped into the data structures of another. Here, we will discuss the issues involved in the mapping and suggest a way in which they may be resolved.

If a new scheduler is going to take over responsibility for path scheduling, then it must have some way of knowing which paths are running. PAVECT specifies the active paths, but the inactive paths may be contained in some arbitrary data structure. It is certainly undesirable for each new scheduler to have to know from which scheduler it is taking over and how that scheduler maintained the set of inactive paths. Hence, we desire a mechanism which will allow a scheduler to be installed without knowledge of the innards of the previous one.

The solution is straightforward. We define the canonical form for the inactive set as follows: the inactive set of paths is in canonical form if and only if all inactive paths are contained on the queue INACTIVEQ. Each procedure that initializes a new scheduler as described in the last section must provide two procedures to be bound as local variables to the names MAPC (map-canonical) and MAPO (map-own.) MAPC is used to map the inactive set from the form used by the scheduler into canonical form; MAPO is used

to map the inactive set from canonical form into the form to be used by the scheduler. MAPC and MAPO are both initially bound in the CI path to the following procedure body:

```
EXPR( ;NONE)NOTHING;
```

since the initial scheduler keeps the inactive set in canonical form. The initializing procedure may map the inactive set from the form being used by the previous scheduler into the form required by the scheduler being initialized by first calling the MAPC procedure associated with the previous scheduler and then calling the MAPO procedure associated with the new scheduler. When control is to be returned to a previous scheduler, then the procedure which has been returned to via the RETFROM("C\I",NIL), i.e. the procedure which had been used to initialize the scheduler, can call its own MAPC procedure and then the previous scheduler's MAPO procedure so that the inactive set may be returned to the form required by the previous one. To illustrate, let us assume that a certain scheduler requires the set of inactive paths be divided into two queues: one queue for those inactive paths which are DORMANT and one queue for those inactive paths which are not. Let us also assume that the procedure used to initialize this scheduler is named INITD and that it takes as argument the ROUTINE to be used as the scheduler. INITD is defined in Figure 2-4. Note that ENTERL, which is defined in Appendix 3, enters the path as the last element

of the queue specified by the second argument.

```

INITD<-EXPR(S:ROUTINE;NONE)
  BEGIN
    MAPC(); NT MAP OLD INACTIVE SET INTO CANONICAL FORM;
    BEGIN
      DECL USER\SCHEDULER:ROUTINE BYREF S;
      DECL INACTD:ARQPTR; NT TO BE USED FOR DORMANT PATHS;
      DECL INACT:ARQPTR; NT TO BE USED FOR OTHER PATHS;
      DECL MAPO,MAPC:ROUTINE;
      MAPO<-EXPR(;NONE)
        BEGIN
          DECL T:ARQPTR BYVAL INACTIVEQ.FIRST;
          DECL Q:ARQPTR;
          TAG:T=NIL => NOTHING;
          Q<-T.NEXT;
          [ ] T.DORMANT => ENTERL(T,INACTD);
            ENTERL(T,INACT) ( ];
          T<-Q;
          GOTO TAG
        END;
      MAPC<-EXPR(;NONE)
        BEGIN
          INACTD.FIRST=NIL => INACTIVEQ<-INACT;
          INACT.FIRST=NIL => INACTIVEQ<-INACTD;
          INACTIVE<-INACT;
          INACTIVE.LAST.NEXT<-INACTD.FIRST;
          INACTIVE.LAST<-INACTD.LAST
        END;
      MAPO(); NT MAP INACTIVE SET INTO NEW FORM;
      C\I(); NT SCHEDULE PATHS WITH NEW SCHEDULER;
      MAPC() NT MAP QUEUES INTO CANONICAL FORM;
    END;
  MAPO() NT MAP QUEUES INTO FORM REQUIRED
    BY OLD SCHEDULER;
END;

```

Figure 2-4: The definition of INITD

4.3 Scheduling Errors

In section 2.3.1, we described the three conditions under which CONTPATH would refuse to pass control to a path. Of these three, two can be explicitly checked by the path scheduler to insure that it does not choose a path which will be rejected by CONTPATH. It is probably desirable, although not absolutely necessary, for the path scheduler to check for these conditions and take appropriate action.* For example, a path which has ELGFLG=FALSE can simply be removed from the inactive set and a path with DORMANT=TRUE can simply be retained in the inactive set.

Recall that a path is being modified if it is active or if an environment modifying control primitive is being applied to it. If it is being modified, the MOD field of its ACTRC has been lSET by a control primitive. Although the scheduler can determine if a path is active by examining the PAVECT, in general it cannot determine if the path is being modified since control primitives can be applied to the path asynchronously with respect to the actions of the CI.

*

Of course, the check for these conditions can be made explicitly in the the body of the scheduler, or implicitly by an error handling routine which responds to the error generated by CONTPATH and returns a different path to be evaluated.

If the scheduler attempts to pass control to an active path, then the path must exist in the active set and the inactive set. Hence, there is either a bug in the scheduler or in one of the procedures which has access to the schedulers queues.

5. EXTERNAL INTERRUPTS

In this section, we will discuss the external interrupt facility of MPEL1. Recall that external interrupts are required to effect processor multiplexing, c.f. 2.3.4. In addition, external interrupts provide a mechanism whereby paths may respond to events which occur outside the scope of the language.

External interrupts affect the evaluation of paths, i.e. the evaluation of the path is interrupted by the occurrence of an interrupt. To be able to speak of one path sending an interrupt to another, it is necessary to extend the concept of external interrupt. This extension will be described below.

5.1 Classes of Interrupts

An interrupt may be loosely defined as a signal which indicates the occurrence of some event. There must be at least two agents associated with an interrupt, namely, one agent to generate the signal and one agent to receive it. It will be convenient to divide interrupts into two classes: external interrupts which are generated by (external) processors, and internal interrupts which are generated by path evaluators.

An internal interrupt is a signal from a path evaluator

to the path it is evaluating. The signal is usually sent to indicate that some error has occurred in the evaluation. The interrupt occurs synchronously with respect to the path's evaluation. For example, if a path attempts to select a non-existent component of a structure, then a signal will be sent to the path to indicate the selection error. In MP_{EL}1, internal interrupts are handled as in EL1 [Weg70]. No change in semantics is necessary for MP_{EL}1 since internal interrupts affect the evaluation of only one path. When an internal interrupt occurs, the identifier environment is searched for a binding of an identifier which is uniquely associated with the interrupt (e.g. "SELECTION\FAULT".) If the identifier is found and it is bound to a procedure definition, then the procedure is called as the path's response to the interrupt. If no binding is found, then a standard system error handling procedure is called.

An external interrupt is a signal which is sent from one processor to another. The signaling processor may either be an evaluator or a special processor which is dedicated to a given task, e.g. a timer, I/O device. The effect of an external interrupt is to interrupt asynchronously the evaluation of the path which is being evaluated by the signaled processor. We may now clarify the distinction between internal and external interrupts. In both cases the final recipient of the interrupt is a path.

In the former case, the signal is generated as a result of some action taken internally by the path itself. In the latter case, the signal is generated by some action which is external to the path. The path is interrupted simply because it is being evaluated by the processor to which the signal was sent.

In EL1, as described in [Weg70], there is only one path of control and only one evaluator, hence, external interrupts may be handled in the same fashion as internal ones. The identifier environment is searched for a procedure which is associated with the particular external interrupt. In MPEL1, however, there are multiple paths of control. Consequently, it is possible that a path is not being evaluated at the time an external interrupt arrives for which it is 'enabled.' In addition, it is usually desirable to associate priorities with external interrupts to facilitate in their processing. EL1 provides no mechanism for treating interrupts on a priority basis. Consequently, additional control apparatus is necessary in order to incorporate external interrupts into the multi-path control structure of MPEL1.

5.2 Interrupt Structure

In this section, we will discuss a number of issues relating to the introduction of external interrupts into

MPEL1. In particular, we will consider the requirements placed upon any interrupt structure by processor multiplexing and multiple paths of control.

External interrupts are associated with processors, not paths. The processor receives the signal and responds to it by interrupting its current activity and taking some pre-specified action. For example, consider the CPU of a digital computer. When an interrupt occurs, the CPU itself is interrupted independent of which process it is executing. Some interrupt program is executed and then the interrupted process is resumed. The interrupt program can inform the process about the occurrence of the interrupt by resuming it at some pre-specified process-dependent location. Thus, if we associate external interrupts with processors, then they can be associated with paths by extension.

Although external interrupts are associated with processors, a language level response to an interrupt must be evaluated in the environment of a path - whichever path is being evaluated by the processor when the interrupt occurs. The response borrows the environment of the current path because it requires an environment for its evaluation and the current path just happens to be available.

Paths may wish to respond to interrupts as well. If we associate responses with interrupts on a path-independent basis, i.e. one response form per interrupt per processor,

then it becomes difficult to allow paths to respond differently to a given interrupt. Conversely, if we associate the response forms on a path-dependent basis, i.e. when an interrupt occurs the response form associated with the current path is used, then it becomes difficult for a path to insure that it will be notified that an interrupt has occurred because of processor multiplexing. It is possible that when an interrupt occurs, the path which is interested in it is not the one which is currently being evaluated.

The essential point is that an interrupt structure is required which will allow a path to be notified of an interrupt even if the interrupt occurs while the associated processor is evaluating another path. In addition, paths must be allowed to respond to interrupts in different ways.

To resolve the issues described above, it would seem desirable to associate an interrupt structure with each processor and, in addition, associate a related structure with each path. The processor level structure may be used to dispatch the interrupt information to all interested paths.

5.3 Processor Level Interrupts

In section 2.3.3, we indicated that the number of processors over which paths are being multiplexed was stored

in NPROC. Let us assume that each of the NPROC processors has NEI associated external interrupts. Each external interrupt has a unique identifier associated with it, e.g. "TIMER", "LIGHT\FEN", "IO\COMPLETION".) In addition, each processor has NPROLEV priority levels, where 1 is the highest priority and NPROLEV is the lowest. A processor may be enabled for one external interrupt at each priority level, but it may not be enabled for the same external interrupt on more than one level. Associated with each external interrupt is a form which is to be evaluated as response to the interrupt.*

A processor may be enabled for an external interrupt by a call to the control primitive ENABLE\PRO.

```
ENABLE\PRO<-CSUBR(EINAME:SYMBOL,LEV:INT,RESP:FORM;NONE)
```

The processor which evaluates this primitive will be enabled for the external interrupt named EINAME at priority level LEV with response form RESP. An error is generated if the processor is already enabled for the interrupt or if the level is already associated with some interrupt.

A processor may be disabled with respect to an external interrupt by calling upon the control primitive DISABLE\PRO.

```
DISABLE\PRO<-CSUBR(EINAME:SYMBOL;NONE)
```

*

The processor level interrupt structure is essentially an abstraction of priority interrupt systems found in practice on contemporary hardware systems.

After a call to `DISABLE\PRO`, the processor is no longer enabled for `EJNAME` interrupts and the level at which it was enabled is available for association with another interrupt.

It is sometimes desirable to modify the response form associated with an interrupt without having to disable and re-enable it. To facilitate this, the data structure which associates response forms with interrupts is accessible from the language, and thus may be modified by assignment. The response forms for all priority levels of all processors are contained in the global data structure `RESPONSE` which is of mode:

```
ROW(NPROC,ROW(NPROLEV,FORM)).
```

Hence, `RESPONSE[N][M]` specifies the response to the interrupt enabled at level `M` on processor `N`. In addition, the trivial control primitives `LEVEL` and `INUSE` may be used to obtain the interrupt status of the current processor. `LEVEL("TIMER")` returns the priority level at which the timer interrupt is enabled. `INUSE(3)` returns the symbolic name of the interrupt enabled at level 3, or `NIL` if the level is not currently associated with an interrupt. Hence, if the "TIMER" interrupt is enabled, then

```
INUSE(LEVEL("TIMER"))="TIMER".
```

Note that the data structures which associate external interrupts with priority levels are not accessible from the language. This restriction is necessary, since correct modification of these structures, i.e. for enabling or

disabling, may require communication with the underlying machine.

The interpretation of an external interrupt is as follows. Let us assume that initially there are no interrupt responses in progress. When an external interrupt occurs, the interrupted processor evaluates the form associated with the interrupt in the environment of the path which it is currently evaluating. If any lower priority interrupts arrive during the evaluation of the response, then the response to the lower level interrupt is not initiated until the higher level response is completed. If a higher level one arrives during the evaluation of the response, then the current evaluation is suspended and the response associated with the higher level one is initiated, i.e. it is nested within the lower level response. Upon completion of the evaluation of a response, the priority of the interrupt associated with the suspended response is compared with the highest priority of the interrupt responses which have not yet been initiated. If the former is greater than or equal to the latter, then evaluation of the suspended response continues. Otherwise, the evaluation of the response associated with the highest level waiting interrupt is initiated. For example, if three interrupts "X", "Y", and "Z" interrupt the processor, and the priorities associated with these interrupts are 3, 1 and 2, respectively, then the arrival of "Y" will suspend the

evaluation of the response for "X". The response for "Y" will not be interrupted by the arrival of "Z". Upon completion of the response for "Y", the response for "Z" will be initiated since it is at a higher priority than "X". Upon completion of the response for "Z", the evaluation of the response for "X" will continue.

If the evaluation of an interrupt response is never completed, then the responses for lower priority interrupts will never be initiated. For example, if an interrupt response performs a CIA which subsequently switches the processor to another path, then the processor will not initiate any lower priority responses because the interrupt nesting is recorded in the intra-path control of the path in which the response was initially evaluated. In addition, if the original path is evaluated by another processor, then an attempt may be made to continue evaluation of a lower priority interrupt while a higher one is in progress. Consequently, although it is desirable for interrupt responses to avail themselves of the power of the CI, they should always return control to the path in which the interrupt originally occurred. ^{*} Hence, processor level interrupt responses may be thought of as 'borrowing' the environment of whichever path is being evaluated at the time

*

An error will be generated by CONTIPATH if an attempt is made to evaluate a different path on the processor, or to evaluate the path on a different processor.

that the interrupt occurs.

If an interrupt response is completed by a RETFROM or a GOTO, then the priority level at which the processor is evaluating is taken to be the priority level of the most recently suspended response above the point to which the RETFROM or GOTO is made, i.e. if the intra-path control is flushed above the point at which an interrupt response was initiated, then the response is automatically completed. Upon completion of the RETFROM or GOTO, evaluation continues with the most recently suspended response, or the highest level waiting response, whichever has the higher priority. We note, however, that although RETFROM and GOTO may be used by response forms, it is probably undesirable to do so. A processor level interrupt response may be responding to an interrupt which is of interest to another path. If the response form is not allowed to complete properly, then the path may never receive the information it desires.

Since the actions of processor level interrupt responses are restricted as described above, an additional mechanism is necessary to make effective use of external interrupts. In particular, we specified that external interrupts would be used to insure that the evaluators were multiplexed across all running paths. But it is not possible to use a processor level interrupt to force the processor to evaluate another path since the evaluation of

the response form must be completed before the switch can occur.* Secondly, we have not specified how a processor level interrupts may be used to dispatch an interrupt to another path. In the next section, we will introduce the additional control apparatus necessary to achieve these capabilities.

5.4 Path Level Interrupts

In this section, we will discuss the concept of path level external interrupts. A path level interrupt is a signal sent from one path to another. Although path level interrupts are external interrupts in the sense that the signal arrives asynchronously with respect to the evaluation of the path, they may be considered to be pseudo interrupts in the sense that the interrupt does not take effect until the path is actually evaluated. Hence, we will refer to path level interrupts as pseudo interrupts.

Each path has NPALEV priority levels associated with it, where 1 is the highest priority and NPALEV is the lowest. A path may be enabled for one pseudo interrupt at each priority level, but it may not be enabled for the same

*

For example, the form CIA(F,MYPATH), where F is a procedure which puts the current path on the INACTIVEQ and sets LASTRUN to NIL (forcing the CI to schedule some other path), could not be used since control leaves the path from within the CIA control primitive and thus the evaluation of the response form is not completed.

pseudo interrupt at more than one level. Associated with each pseudo interrupt is a form which is to be evaluated as response to the interrupt. Thus, the path level pseudo interrupt structure parallels the processor level 'real' interrupt structure. A pseudo interrupt is referenced by a symbolic name, e.g. "WALDO", "PLEASE\TERMINATE". A number of control primitives are defined to effect enabling, disabling, masking and generation of pseudo interrupts. Their description follows.

The last argument to each of the control primitives described below specifies the path to which the actions of the primitive are to be applied. If the argument is not supplied to the primitive, i.e defaulted to NIL, then the actions are to be applied to the path which executes the call. If the path to which the primitive applies is not the current path, then it must not be modified while the primitive is being applied. Consequently, the path cannot be active.

The control primitive ENABLE\PATH enables a path for a pseudo interrupt.

```
ENABLE\PATH<-CSUBR(PEINAME:SYMBOL,LEV:INT,
                  RESP:FORM,PATH:ARPTR;NONE)
```

The path is enabled for pseudo interrupt PEINAME at level LEV with response form RESP. An error is generated if the path is already enabled for a pseudo interrupt at level LEV

or if the path is already enabled for PEINAME interrupts at some level.

A path may be disabled with respect to a pseudo interrupt by calling upon the control primitive `DISABLE\PATH`.

`DISABLE\PATH<-CSUBR(PEINAME:SYMBOL,PATH:ARPTR;NONE)`

After a call to `DISABLE\PATH`, the path is no longer enabled for the interrupt specified by PEINAME and the level at which PEINAME was enabled is available for association with another pseudo interrupt.

A pseudo interrupt may be generated by a call to `INTERRUPT`.

`INTERRUPT<-CSUBR(PEINAME:SYMBOL,P:ARPTR;NONE)`

Let us first assume that P is not the path which called `INTERRUPT`. Recall that P cannot be active. If the path specified by P is enabled for PEINAME interrupts at some level, then the response form that it has associated with PEINAME will be evaluated in its environment as soon as it is evaluated by some processor. If P was in the midst of the evaluation of a response to some pseudo interrupt at a higher priority level, then the response to PEINAME will be evaluated when all higher priority responses have been completed. The interrupt is only 'pseudo' since no processor is physically interrupted. `INTERRUPT` merely records information in the path's activation record. To

send a pseudo interrupt to an active path, it is necessary to physically interrupt the processor which is evaluating the path and then use the primitive INTERRUPT. Hence, a 'real' external interrupt is required. The control primitive which provides this facility is described in the next section.

If P specifies the path which has called INTERRUPT, then the response to the interrupt is immediately evaluated in the current path's environment, unless the path is currently evaluating a higher level interrupt response.

The interpretation of path level interrupts is identical to that of processor level interrupts, as described in the last section. Lower priority level responses are delayed until higher level ones complete. Higher level responses take precedence over the evaluation of lower level ones. However, we have not specified the relation between processor interrupt levels and path interrupt levels. The relation is as follows: the path interrupt levels are of strictly lower priority than processor interrupt levels. Hence, any processor interrupt takes precedence over any path level interrupt. Consequently, if a processor level response generates path level interrupts for the current path, then the path level responses will not be initiated until all processor level responses have been completed.

The addition of the path interrupt structure resolves the problems described at the end of the last section. First, to force processor multiplexing, the external interrupt can generate a pseudo one which will be processed after all processor level responses are completed. The response to the pseudo interrupt can safely switch the processor to another path via a CIA call. The original path is simply left in the midst of a path level interrupt response. Secondly, a RETFROM or GOTO out of a path level response can only affect the current path's processing. Hence, if a processor level interrupt desires to perform a RETFROM or GOTO without any effect upon a lower priority processor interrupt, then it can generate a path level one to perform the desired action. Finally, the processor level interrupt response may dispatch the fact that a given interrupt has occurred by sending pseudo interrupts to all interested paths.

It is sometimes desirable to mask a path against certain pseudo interrupts, i.e. a path may wish to remain enabled for a given interrupt, but have the interrupt response delayed for some time. Hence, a mechanism is

*

Note that this cannot be achieved by having the path generate a self pseudo interrupt at a priority higher than the interrupt to be masked, since all lower level interrupts will then be masked as well, which is not necessarily the desired effect.

desired which will mask a path against an interrupt while allowing the occurrence of other interrupts at higher or lower priority levels. Two control primitives provide this facility: MASK and UNMASK.

```
MASK<-CSUBR(PEINAME:SYMBOL,PATH:ARPTR;NONE)
```

```
UNMASK<-CSUBR(PEINAME:SYMBOL,PATH:ARPTR;NONE)
```

If a pseudo interrupt is sent to a path which has masked against that interrupt, then the fact that the interrupt occurred is recorded, but no response is generated. If an interrupt is UNMASKed and if the interrupt occurred while it was masked, then the response is generated according to the priority rules described above.

The data structures associated with path level interrupts are stored in the activation record of each path. Hence, they are accessible from the language. As was the case with processor level interrupts, the response forms associated with pseudo interrupts may be directly modified without disabling and re-enabling the interrupt. Note, however, that direct modification of other structures may not have the desired effect. For example, if a path

*

Multiple interrupts are lost. Alternatively, we could maintain a count of the number of times a given pseudo interrupt has occurred. An interrupt structure of this sort would be a straightforward extension of the current facility.

**

The structures are described in detail in section 4.3.13.

attempts to unmask itself, with respect to a certain interrupt, by direct modification of the appropriate data structure, then the response form will not be automatically triggered. Hence, the structures may be examined to discern the status of a path's interrupt levels, but most modifications should be made via the appropriate control primitive.

5.5 Relation to Processor Multiplexing

In section 2.3.4, we mentioned the use of interrupts in processor multiplexing but deferred explanation; we now remedy this omission. There are two problems to be solved. First, how can an idling processor be assigned to an inactive path? Second, how can an active path occasionally be forced to perform a CIA call which will give its processor to another path? We will assume that the external interrupts "TIMER" and "PRO\PRO" are associated with each processor. A "TIMER" interrupt is sent to its associated processor after a fixed interval of time has elapsed. Hence, a processor may keep track of how long it has been evaluating a particular path. A "PRO\PRO" interrupt is one which is sent from one processor to another. We will assume that each processor is able to send a "PRO\PRO" interrupt to any other processor. A "PRO\PRO" interrupt may be used to force a processor to stop evaluating a particular path.

The control primitive STOP\PATH may be used to send a "PRO\PRO" interrupt to a processor.

```
STOP\PATH<-CSUBR(P:ARPTR;NONE)
```

STOP\PATH sends a "PRO\PRO" interrupt to the processor which is evaluating path P. If P is not currently being evaluated, then no action occurs. STOP\PATH may only be called from the environment of the CI. This restriction is necessary since the assignment of processors to paths can be unambiguously determined only by the processor evaluating the CI

We will assume that each processor is enabled for "TIMER" and "PRO\PRO" interrupts, viz.

```
ENABLE\PRO("PRO\PRO",1,PRO\PRO\FORM)
```

```
ENABLE\PRO("TIMER",2,TIMER\FORM)
```

In addition, we will assume that GET\PATH enables each path P for the following pseudo interrupts.

```
ENABLE\PATH("CI\TO\PATH",1,CI\PATH\FORM,P)
```

```
ENABLE\PATH("TIME\OUT",2,TIME\OUT\FORM,P)
```

The four forms are given in Appendix 3. Their use in processor multiplexing will be described informally below.

In section 2.3.3, we postponed discussion of the structure PIVECT, which is defined in the environment of the control interpreter. Here, we describe its use in the processing of "PRO\PRO" interrupts. The PIVECT is used essentially as a communication vector to allow one processor

to specify a list of actions (forms to be evaluated) to be taken by another processor, where $PIVECT[N]$ is the list for the N th processor.

For example, let us assume that the scheduler wishes to interrupt an idling processor because some real path requires evaluation. Let us also assume that the idling path for the processor is P , i.e. $PAVECT[N].IDLEPATH=P$, where N is the number of the processor. The scheduler executes $STOP\PATH(P)$. A "PRO\PRO" interrupt is sent to the idling processor. The response form $(PRO\PRO\FORM)$ generates a pseudo interrupt "CI\TO \PATH" for path P . The response form $CI\PATH\FORM$ passes control to the CI and evaluates all forms on the list $PIVECT[PROCNUM]$. In particular, if the scheduler has previously placed a form on the list which when evaluated will set $LASTRUN$ to NIL , then upon completion of the CIA call the processor will be assigned to an inactive path. Hence, the effect of the above scenario is to force an idling processor to pass control to the CI where the scheduler can assign it to an inactive path. Note that the pseudo interrupt is necessary since the processor is to be switched to another path. All processor level responses must be completed before the switch can be made.

$STOP\PATH$ may also be used in conjunction with $PIVECT$ to force an active path to cease evaluation. An appropriate

form can be put on the PIVECT list of the processor which is evaluating the path and then a call to STOP\PATH can be executed. When control passes to the CI due to the response to the "CI\TO\PATH" interrupt, then the evaluation of the form on the PIVECT list can take the desired action. For example, examine the definition of the procedure SUSPEND in section 3.3. If path P wishes to suspend an active path Q, then P adds a form to the PIVECT entry for the processor of Q and then executes STOP\PATH(Q). When the form is evaluated, Q will be suspended and P can be allowed to resume execution.

It is important to note that if a "PRO\PRO" interrupt is sent to the processor of a path which is waiting to perform a CIA call, then the interrupt response will be evaluated before the CIA call is executed.^{*} Consequently, it is impossible for the processor to be switched to another path before the interrupt response is generated. Thus, the response will always be generated in the environment of the path specified by STOP\PATH.

We now turn to the problem of how to effect the

*

This is not true if the processor is enabled for "PRO\PRO" interrupts at some lower priority and the CIA call in question is executed by the response to a higher level processor interrupt. In this case, however, it is still impossible for the processor to be switched to another path before the response is evaluated due to the constraints upon processor level interrupts, c.f. 2.5.3.

multiplexing of processors over all paths. The "TIMER" interrupt provides a straightforward solution. Whenever a timer interrupt occurs, the associated response form decrements a count stored in the activation record of the path that it is evaluating. When the count reaches zero, the pseudo interrupt "TIME\OUT" is generated. The response to the pseudo interrupt performs a CIA call which places the path at the end of the INACTIVEQ and sets LASTRUN to NIL to indicate that a new path should be scheduled to run on the processor. The count is stored in the integer component TICKS\LEFT and is initialized by the scheduler to be the number of timer 'ticks' which may occur before the path is forced to 'time-out.' As was the case with "PRO\PRO" interrupts, a pseudo interrupt is required since the processor level response must be completed before the processor can be assigned to another path.

5.6 Data Passage

In the previous sections, we have neglected to discuss the fact that an external interrupt may have data associated with it. For example, a light-pen interrupt may specify spacial coordinates. We will assume that the information associated with a given external interrupt will be stored as the value of an associated global variable. As the data is completely dependent upon the type of external interrupt and upon the implementation of the language, e.g. how I/O

transactions are specified, we will not specify the names or the modes of these variables.

The primary issue with respect to external interrupt data is whether or not there is a mechanism which assures that the data structures involved can be updated safely. For example, suppose that an external interrupt "FOO" has associated with it some data D. Whenever a "FOO" interrupt occurs, D is to be added to a list of Ds which are to be processed by a path P. After P processes an element of the list, it removes the element and processes the next one. Some synchronization is required to insure that the list is updated safely. The solution is straightforward. The response to the "FOO" interrupt performs a CIA call to add D to the queue and then sends a pseudo interrupt to path P to indicate that additional data has arrived. P also performs a CIA call to remove elements from the list. Hence, the list will be safely updated since only one path at a time can pass control to the CI.

There is one hitch, however! Suppose that the "FOO" interrupt occurs on the processor which is evaluating the CI path during the performance of the CIA called procedure which removes elements from the queue. How can the information be added safely to the list? Many solutions are possible. The simplest one seems to be as follows. If P has passed control to the CI to delete an element from the

list, then P is not currently being evaluated by a processor. The interrupt response can detect this case by using MYPATH and by checking the value of LASTRUN. D can be stored directly into path P using one of the control primitives described earlier, e.g. PSTORE. When P resumes evaluation, it can detect that the data was stored in its environment while the CIA call was being evaluated, and thus, process it directly.

6. INDEX TO CHAPTER 2

CIA	41
CIA\ARG	42
CIA\FN	42
CIA\RESULT	43
CI\PATH\FORM	83
CLEAR	31
CONTPATH	44
COPY	39
DELETE\PATH	15
DEPENV	34
DISABLE\PATH	78
DISABLE\PRO	71
DORMANT	24
DPAP	34
ELGFLG	15
ENABLE\PATH	77
ENABLE\PRO	71
EVAL	40
GET\PATH	15
GOTO	37
INACTIVEQ	48
INTERRUPT	78
IN\USE	72

LASTRUN	48
LEVEL	72
MASK	81
MDEF	33
MOD	45
MYPATH	39
NFPROC	49
NPROC	48
PAP	17
PAPQ	17
PAVECT	49
PCIAR	41
PFETCH	26
PIVECT	50, 83
PROCNUM	49
PRO\PRO\FORM	83
PSTORE	26
RESPONSE	72
RETFROM	38
RUNSET\FLAG	50
STKEFLG	15
STOP\PATH	83
TERMINATION\FORM	28

TICKS\LEFT	86
TIMER\FORM	83
TIME\OUT\FORM	83
TSET	31
UNMASK	81
USER\SCHEDULER	49

Chapter 3

EXTENSIONS

In this chapter, we illustrate by example how the primitives and framework of MPEL1 can be used to synthesize a wide variety of multi-path control structures. The examples range in complexity and familiarity from coroutines to relatively continuous evaluation. This chapter serves two purposes. First, it reinforces the reader's understanding of the multi-path facility by presenting examples which have appeared frequently in the literature. Thus, it serves as a supplement to the informal description of Chapter 2. Second, it demonstrates the power of the facility for both the implementation and clarification of complex control structures.

In each of the sections below, the desired multi-path behavior is described informally and then a set of MPEL1 procedures which effect the control structure are presented. These procedures, described in terms of the control primitives, may be viewed as defining extensions to MPEL1 to allow for the specified control structure. In some sections, we have included a programming example to

*

Of course, these extensions appear syntactically as procedure calls. More convenient notations can be realized through the use of a syntax-extension facility, c.f. 1.1.1.

illustrate how the extension might be used in practice.

All of the examples in this chapter assume that the control interpreter is being driven by the procedures of Appendix 3. Hence, the CI environment is as described in section 2.3.3.

1. COROUTINES

A set of paths exhibit a coroutine relationship if only one path from the set is being evaluated at any given time, c.f. 1.2.1, 2.1.3. The active path may 'resume' another path, which implies that control is to be transferred from the former to the latter leaving the evaluation state of the former intact. Evaluation of the latter path proceeds from the point it was at the last time it was active. Here, we will define the control functions COCALL, which is used to initialize a coroutine path, and RESUME, which is used to transfer control between coroutine paths, and demonstrate their use in solving a simple problem.

COCALL takes as argument a procedure call to be evaluated in a new path. It creates a new path, uses PAP to set up the procedure call and a dummy call to RESUME, and returns a pointer to the new path.

```

COCALL <- EXPR(COCALLP:FORM UNEVAL; ARPTR)
BEGIN
  DECL P:ARPTR BYVAL GET\PATH(1);
  PAP(COCALLP,P);
  PAPQ(RESUME(NIL,NIL),P);
  NT Dummy call to RESUME for first
    resumption;
  P
END;

```

RESUME takes two arguments. The first specifies the path (PATH) to which control is to be passed. The second specifies the value (V) to be returned from the call to RESUME contained in the environment of PATH. RESUME PAP's a call to RETFROM into the environment of PATH. The procedure to be returned from is RESUME, i.e. the call to RESUME in the environment of PATH, and the value to be returned is V. RESUME then transfers control to PATH by CIA calling an explicit procedure which simply sets LASTRUN to PATH so that when control leaves the CI, PATH will be evaluated instead of the original path. When PATH is evaluated, the RETFROM is executed and V is returned as the value of the call to RESUME in the environment of PATH. Note that the original path is left in a state such that when another path tries to resume it, then the call to RESUME just described is the one which will be returned from. Also note that the first time a COCALLED path is resumed a return is made from the dummy call to RESUME and then the COCALLED procedure call is evaluated.


```

RESUME <- EXPR(PATH:ARPTR, V:ANY; ANY)
  BEGIN
    PAPQ(RETFROM("RESUME",V),PATH);
    CIA(EXPR(P:ARPTR; NONE)(LASTRUN<-P),PATH)
  END;

```

Consider the following problem: given two binary trees x and y , where x and y have the same number of nodes but not necessarily the same structure, walk each tree in prefix order and assign to each node of y two times the node value of the corresponding node of x . E.g.

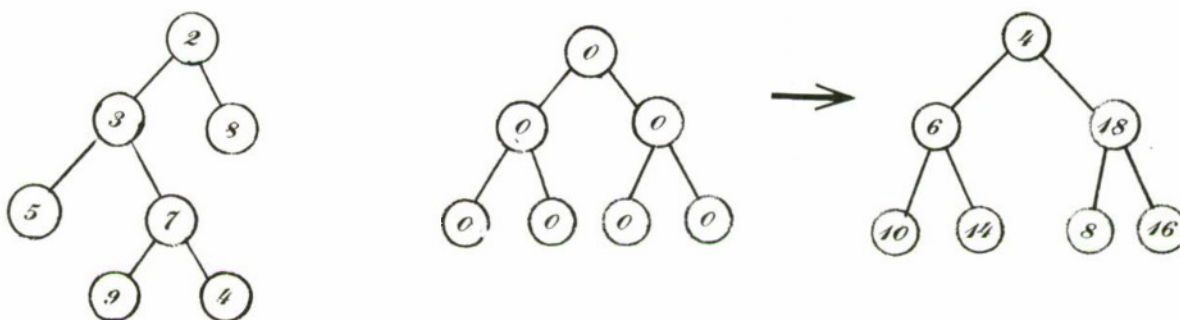


Figure 3-1. Trees x , y and modified y

The data structure definitions are:

```
TREE <- PTR(NODE);
```

```
NODE <- STRUCT(LS:TREE, RS:TREE, NODE\VAL:INT);
```

To solve this problem, we will define a procedure `TREE\DOUBLE` which will create two new paths P_x and P_y (using `COCALL`) making a total of three paths, including the path in which `TREE\DOUBLE` is called (which we will refer to as P_o .) P_x (P_y) will perform the prefix walk of tree x (tree y .) P_o will `RESUME` P_x (P_y) when it requires the next node of tree x

(tree y). When RESUMEd, Px (Py) will walk to the next node of the tree and then RESUME Po, passing it a pointer to the node. Note that since Px and Py are separate paths they retain their internal state upon returning the next node to Po.

TREE\DOUBLE is defined as follows.

```
TREE\DOUBLE <- EXPR(X:TREE, Y:TREE; TREE)
  BEGIN
    DFCL PX, PY:ARPTR;
    DFCL NX, NY:TREE
    PX <- COCALL(WALK(X, MYPATH));
    PY <- COCALL(WALK(Y, MYPATH));
    NT Create the paths Px and Py
      and set up calls to WALK
      in their environments;
  LOOP: NX <- RESUME(PX, NIL);
    NY <- RESUME(PY, NIL);
    NT Resume Px to get the next
      node of x, which is then
      bound to NX. The result of
      RESUME is a pointer to the
      next node. Do the same
      for Py;
    NX=NIL => Y;
    NT Px returns NIL
      when all nodes have been
      walked;
    VAL(NY).NODE\VAL <- 2*VAL(NX).NODE\VAL;
    NT Make the node of y be two
      times the node of x;
    GOTO LOOP
  END;
```

The procedure WALK is defined as follows:

```
WALK <- EXPR(T:TREE, COPATH:ARPTR; NONE)
  BEGIN
    WALK1(T); NT Walk the tree;
    RESUME(COPATH, NIL)
    NT Resume Po with NIL to indicate
      that all nodes have been processed;
  END;
```

where

```

WALK1 <- EXPR(T:TREE; NONE)
  BEGIN
    T=NIL => NOTHING;
    ' RESUME('COPATH,T); NT Resume Po with T;
    WALK1(T.IS);      NT Walk the tree via
                      recursive calls;
    WALK1(T.RS)
  END;

```

2. SYNCHRONIZATION

In section 1.2.1, we indicated that semaphores and their associated operators (P and V) may be used to synchronize parallel processes, but deferred explanation. Here, we will show how P and V may be defined in MPPL1.

As described by Dijkstra [Di68a], a semaphore is a 'special-purpose' integer upon which only two operations are valid - P and V. The V operation increases the value of the semaphore by 1 in a single indivisible operation. The P operation decreases the value of a semaphore by 1 as soon as the resulting value would be non-negative (≥ 0). Hence, a P operation on a non-positive (≤ 0) semaphore cannot be completed until another process performs a V operation on the same semaphore. The P operation, therefore, represents a potential delay in the execution of a process. Note that if some N processes all perform a P operation on a semaphore whose value is zero, and some other process performs a V on it, then only one of the N processes will be allowed to proceed.

In MPEL1, a semaphore may be defined as a pointer to a STRUCT consisting of two components, viz.

```
SEM <- PTR(SEM/ELT);
SEM\ELT <- STRUCT(COUNT:INT,WLIST:ARQPTR);*
```

The first component COUNT is an integer that specifies the semaphore's value. The second component is a queue of ACTRCs (linked together through their NEXT components) which corresponds to those paths which have started, but not yet completed, P operations on the semaphore. Hence, the WLIST holds all paths whose progress has been delayed due to the non-positivity of the semaphore. Here, the indivisibility of P and V is effected by performing the operations in the environment of the CI, where the data structures can be safely modified. The MPEL1 definitions of P and V are as follows:

```
P<-EXPR(X:SEM; NONE)
  BEGIN
    DECL Y:SEM\ELT BYREF VAL(X);
    MYPATH # PCIAR => CIA("P",X);
    Y.COUNT GT 0 => Y.COUNT <- Y.COUNT-1;
    ENTERL(LASTRUN, Y.WLIST);
    LASTRUN <- NIL
  END;
```

```
V <- EXPR(X:SEM; NONE)
  BEGIN
    DECL Z:ARPTR;
```

* The mode ARQPTR is defined as a STRUCT(FIRST:ARPTR, LAST:ARPTR), c.f. 3.3.3.

** Recall that PCIAR points to the CI's ACTRC and that ENTERL adds the path which is its first argument to the end of the queue specified by its second argument, c.f. Appendix 3.

```

DECL Y:SEM\ELT BYREF VAL(X);
MYPATH # PCIAR => CIA("V",X);
Y.COUNT <- Y.COUNT+1;
Y.WLIST.FIRST=NIL => NOTHING;

```

NT Complete the P operation for
one delayed path;

```

Y.COUNT <- Y.COUNT-1;
Z <- Y.WLIST.FIRST;
Y.WLIST.FIRST <- Y.WLIST.FIRST.NEXT;
Y.WLIST.FIRST=NIL -> Y.WLIST.LAST <- NIL;
ENTERL(Z, INACTIVEQ);
RUNSET\FLAG <- TRUE

```

END;

A P operation is realized by CIA calling the same procedure P with the SEM as argument. Hence, if a path executes P(S), P(S) is also executed in the CI. Here, if the count is positive, then it is decremented by one and the path is allowed to continue. Otherwise, the path is entered on the semaphore's WLIST and LASTRUN is set to NIL to indicate that the path cannot proceed. The scheduler will choose some other path to run. Thus, the P operation performed by the path is not allowed to 'complete.'

The V operation causes V(S) to be applied in the CI environment. Here, the count is incremented by one. If the WLIST is empty, then no further action is taken. Otherwise, the procedure 'completes' the delayed P operation for some path on the WLIST by decrementing the count, removing the path from the list, adding the path to the inactive set so that it will be scheduled, and setting RUNSET\FLAG to TRUE to indicate to the scheduler that there exist additional paths to be run.

If a semaphore, say S , is initialized to 1, then it may be used to control a single-access resource, provided that all paths perform a $P(S)$ before accessing the resource and a $V(S)$ upon completion. Hence, once a path has performed a $P(S)$ all others will be prevented from accessing the resource until the corresponding $V(S)$ is performed. If, however, a semaphore S is initialized to some $n > 1$, then n processes may perform $P(S)$ before the semaphore becomes non-positive. A semaphore initialized in this way can be used to represent the n -fold availability of a resource, (e.g. n tape drives.)

Saul and Riddle [Sa71] have shown that if P and V are allowed to return values, then the number of different semaphores and the number of references to semaphores in a program can be reduced. For example, a list of free 'buffers' could be associated with a semaphore and the semaphore initialized to the number of buffers. The P operation removes one buffer from the list and returns a pointer to the buffer as its value. The V operation takes an additional argument which is a pointer to a buffer to be returned to the free list. Hence, a single extended semaphore is used both to maintain a count of the free buffers and to synchronize access to a free list as opposed to having one semaphore maintain the count and a second synchronize access. The definition of extended semaphores, as described above, is straightforward in MPEL1. The mode

SEM\ELT can be modified to include a pointer to the list and the definition of P (V) can be changed to remove (return) buffers to the list while control resides in CI.

Variations on P and V, such as wait and cause in OREGANO [Be71], can be constructed in MPEL1 by defining the appropriate data structures and then using the CI to provide indivisible operation coupled with a mechanism for indicating that a previously blocked path may now continue execution.

3. PARALLEL PROCESSING

In this section we will discuss a set of procedures which may be used to ~~manage~~ ^{*} paths which are being evaluated concurrently, i.e. as asynchronous processes. These include procedures which allow for the creation, synchronization, suspension, and termination of parallel paths.

For this section we will assume that the following components have been added to the definition of ACTRC, i.e. they are extended components, c.f. 2.2.1.

```
ACTRC<-STRUCT( ...
               PAL:STRUCT(OWNER:ARPTR,WLIST:ARQPTR),
```

*

If two paths are to be evaluated concurrently, then they may be evaluated simultaneously, sequentially in any order, or in an interleaved fashion. In particular, no assumption is made as to the relative speeds of (the evaluations of) the paths. This assumption is consistent with standard definitions of parallel processing [Di68a].

```
PVALRET:BOOL,PVALQ:ARQPTR,PAVAL:REF, ...);
```

PAL is a path-access-lock which is used to synchronize access to the activation record and environment of the path. PVALRET is TRUE if and only if the path has returned a value. PVALQ is a queue of paths waiting for the path to return a value. PAVAL is a pointer to the 'value' of the path. These components will be discussed in more detail below.

The procedure CREATE takes a single argument which is a procedure call to be evaluated asynchronously with respect to the current path. CREATE allocates a new path, PAPS the procedure call into its environment and then enters the path on the INACTIVEQ so that it will be assigned to a processor. Note that RUNSET\FLAG is set to TRUE to inform the scheduler that additional paths may be scheduled, c.f. 2.3.3. The TERMINATION\FORM will be discussed later in this section. The current path and the CREATED path are evaluated concurrently.

```
CREATE<-EXPR(CREATEF:FORM UNEVAL; ARPTR)
  BEGIN
    DECL CREATEP:ARPTR;
    CREATEP <- PAP(CREATEF, GET\PATH(1));
    CREATEP.TERMINATION\FORM <-
      QUOTE(TERMV(LAST\VALUE,MYPATH));
    NT See the discussion of TERMV below;
    ENTER\INACTIVEQ(CREATEP);
    CREATEP
  END;
```

```
ENTER\INACTIVEQ(P:ARPTR;NONE)
  BEGIN
    MYPATH#PCIR => CIA("ENTER\INACTIVEQ", P);
```

```

ENTERL(P, INACTIVEQ);
RUNSET\FLAG <- TRUE
END;

```

The procedure RESUME, which is used to transfer control between coroutines, uses the primitive PAP without determining if the path PAPed into is active or not. RESUME assumes that the path is not active. This assumption is reasonable since in a coroutine relationship only one path is active at a time. With asynchronous paths, however, the above assumption is not valid - two paths may be active simultaneously. If a path attempts to PAP a procedure into the environment of an active path, then an error occurs, c.f. 1.1.3. An error also occurs if two paths simultaneously attempt to PAP procedures into the environment of a third. In general, a mechanism is required which allows one path to examine and modify another one without interference from any other concurrent path. In addition, a control function is required which will force a path to become not active and prevent it from becoming active for some period of time.

We could associate a binary semaphore, i.e. one whose COUNT is initialized to 1, with each path to synchronize access. When one path wishes to access another, then it performs a P operation on the path's semaphore, modifies the path, and then performs a V operation to release the path. In the last section, however, we observed that extended semaphores are sometimes more convenient than pure ones.

Here, although semaphores are sufficient to provide the desired synchronization, they are deficient in one respect: the semaphore does not specify who has access to the path. It only records the fact that some path is accessing it. The lack of information can be inconvenient in certain situations. For example, if a path P0 passes a path P1 to some procedure f, viz.

f(p1)

and f must modify the environment of P1, then f has no way of knowing whether or not P0 already has access to the path. In particular, if P0 has access to P1, having performed a P on the appropriate semaphore, and f performs a P on the same semaphore, then P0 will be permanently blocked. Hence, f would have to take a second argument which specifies whether or not P0 already has access to P1.

In lieu of P and V, we will use the procedures LOCKP and UNLOCKP to synchronize access to a path. LOCKP takes an ARPTR as argument and returns FALSE if the path executing LOCKP already has access to the path. Otherwise, it returns TRUE as soon as the path may have access. Hence, a procedure can determine whether or not the path in which it is called already had access at the time it was called. UNLOCKP allows some other path to have access to the path which is the argument to UNLOCKP. LOCKP and UNLOCKP use the PAL field of the activation record which is a structure that consists of an OWNER field and a WLIST (as in a SEM\ELT.)

The OWNER field is either NIL (if no path has access) or is the ARPTR of the path which has access. LOCKP and UNLOCKP are defined as follows.

```
LOCKP<- EXPR(P:ARPTR; BOOL)
  BEGIN
    DECL OWNER:ARPTR BYREF P.PAL.OWNER;
    MYPATH#PCIAR =>
      BEGIN
        OWNER=MYPATH => FALSE;
        NT Already locked by this path;
        CIA("LOCKP", P);
        TRUE
      END;
    BEGIN
      OWNER=NIL => OWNER<-LASTRUN;
      ENTERL(LASTRUN,P.PAL.WLIST);
      LASTRUN <- NIL
    END;
    TRUE
  END;
```

```
UNLOCKP <- EXPR(P:ARPTR; NONE)
  BEGIN
    DECL OWNER:ARPTR BYREF P.PAL.OWNER;
    DECL Q:ARPTR;
    MYPATH # PCIAR =>
      BEGIN
        OWNER=NIL => NOTHING;
        CIA("UNLOCKP",P)
      END;
    (Q<-P.PAL.WLIST.FIRST)#NIL =>
      BEGIN
        REMOVE(Q,P.PAL.WLIST);
        NT c.f. Appendix 3;
        ENTERL(Q, INACTIVEQ);
        NT Allow path to be scheduled;
        RUNSET\FLAG <- TRUE
        P.PAL.OWNER <- Q;
        NT Q has access to P;
      END;
    OWNER <- NIL
    NT No paths waiting;
  END;
```

Once a path P0 has LOCKPed another, say P1, no other path can access P1. Before P0 can modify P1, however, it must insure that P1 is not active and that it will not

become active while being modified. The procedure SUSPEND may be used to achieve this effect. If the path to be suspended is not active, then SUSPEND simply sets the DORMANT field of its ACTRC to TRUE to indicate to the scheduler that the path should not be scheduled, c.f. 2.2.4. If the path is active, then SUSPEND sends a "PRO\PRO" interrupt to the processor which is evaluating the path and adds a form to the appropriate PIVECT entry. The form will be evaluated when the processor transfers control to the CI due to the "CI\TO\PATH" pseudo interrupt, c.f. 2.5.5. The evaluation of the form causes the DORMANT field of the path being suspended to be set to TRUE, and allows the suspending path to continue execution.

```

SUSPEND <- EXPR(P:ARPTR; NONE)
  BEGIN
    DECL B:BOOL;
    DECL Q:ARPTR;
    DECL PROC:INT;
    MYPATH # PCIAR =>
      BEGIN
        B <- LOCKP(P);
        NOT P.DORMANT -> CIA("SUSPEND",P);
        B -> UNLOCKP(P)
      END;
    LASTRUN=P =>
      BEGIN
        P.DORMANT <- TRUE;
        ENTER\INACTIVEQ(P);
        UNLOCKP(P);
        LASTRUN <- NIL
      END;

    NT Determine if P is active

    FOR I <- 1,..., NPROC TILL PROC>0 DO
      [ ] PAVECT[I].CURPATH = P => PROC<-I ( );
    PROC = 0 => P.DORMANT<-TRUE;
    NT P is not active;
    PIVECT[PROC] <-
      CONS(LIST("SUSPREQ",

```



```

        ALLOC(REF LIKE
              ALLOC(ARPTR LIKE LASTRUN))),
        PIVECT[PROC]);
NT The form is (SUSPREQ P), where P
  is the suspending path;
STOP\PATH(P);
LASTRUN <- NIL
END;

```

```

SUSPREQ <- EXPR(P:ARPTR; NONE)
BEGIN
  ENTER\INACTIVEQ(P);
  LASTRUN.DORMANT <- TRUE;
  ENTER\INACTIVEQ(LASTRUN);
  LASTRUN <- NIL
END;

```

```

CONS <- EXPR(A:FORM, B:FORM; FORM) ALLOC(DTPR OF A,B);
LIST <- EXPR(F:FORM LISTED; FORM) (LIST1(F));
LIST1 <- EXPR(F:FORM; FORM)
BEGIN
  F = NIL => NIL;
  CONS(EVAL(F.CAR),LIST1(F.CDR))
END;

```

If the path (P0) calling SUSPEND has already LOCKPed the path to be suspended (P1) and P0#P1, then upon return from SUSPEND P0 still has access to the suspended path. If, however, P0=P1 (self-suspension,) then SUSPEND will UNLOCKP the path.

The procedure CONTINUE allows a SUSPENDED path to continue execution. CONTINUE simply sets the DORMANT field of the ACTRC to FALSE. If the path is on the INACTIVEQ then it will be scheduled as usual. Otherwise, it will only continue execution when it is put on the INACTIVEQ. CONTINUE leaves the PAL in the state it was in when CONTINUE

was called. CONTINUE returns TRUE or FALSE as the path was or was not suspended.

```

CONTINUE <- EXPR(P:ARPTR; BOOL)
  BEGIN
    DECL R, B:BOOL
    MYPATH#PCIR =>
      BEGIN
        R <-
          BEGIN
            B <- LOCKP(P);
            NOT P.DORMANT => FALSE;
            CIA("CONTINUE",P);
            TRUE
          END;
        B -> UNLOCKP(P);
        R
      END;
    P.DORMANT <- FALSE
  END;

```

Using the control procedures described above, we can now define PAPPLY - a procedure which can be used to PAP a procedure call into the environment of a concurrent path. PAPPLY uses LOCKP to obtain access to the path. It then SUSPENDs the path if necessary, PAPs the procedure call, and then allows the path to continue execution if it was not suspended previously.

```

PAPPLY <- EXPR(PAPPLYF:FORM UNEVAL; PAPPLYP:ARPTR; ARPTR)
  BEGIN
    DECL PAPLYB, PAPLYSB:BOOL;
    PAPLYB <- LOCKP(PAPPLYP);
    NOT PAPPLYP.DORMANT ->
      BEGIN
        PAPPLSB <- TRUE;
        SUSPEND(PAPPLYP)
      END;
    PAP(PAPPLYF, PAPPLYP);
    PAPLYSB -> CONTINUE(PAPPLYP);
    PAPLYB -> UNLOCKP(PAPPLYP)
  END;

```

It is sometimes useful for a path to 'return' a value.

For example, one path might CREATE a set of paths to be evaluated concurrently, wait for all of them to terminate and then use the values computed by the paths. The procedures WAITV and TERMV, in conjunction with the ACTRC components PVALRET, PVALQ, PAVAL, can be used to wait for a path's value and to specify the value to be returned by a path upon termination, respectively.

WAITV examines the PVALRET component of the path X. If it is TRUE, then the path has terminated and the value associated with the path, which is referenced by the PAVAL component, is returned immediately. Otherwise, X has not yet terminated. In this case, the current path is queued on the PVALQ of X to indicate that it is waiting for X's value and LASTRUN is set to NIL to indicate that the current path is blocked. WAITV uses LOCKP to insure that X does not terminate while it is examining the ACTRC.

```

WAITV <- EXPR(X:ARPTR;REF)
  BEGIN
    DECL Y:REF;
    DECL B:BOOL;
    MYPATH # PCIR =>
      BEGIN
        L: B<-LOCKP(X);
          X.PVALRET =>
            BEGIN
              Y<-X.PAVAL;
              B->UNLOCKP(X);
              Y
            END;
          CIA("WAITV",X);
          GOTO L
        END;
      ENTERL(LASTRUN,X.PVALQ);
      NT Add the path to the queue of paths
        waiting for the value of X;
      UNLOCKP(X);

```



```

        NT Allow other paths to access X;
        LASTRUN <- NIL
    END;

```

TERMV can be used to specify that a path P is to be terminated and that a value V is to be the value associated with the path. TERMV sets PVALRET to TRUE to indicate that a value has been returned. If the mode of V is not of class PTR, then V is copied into the heap and the value of the path is a pointer to the copy. In any case, a pointer to the path's value is stored in PAVAL. All paths waiting for the value (and termination) of the path are added to the INACTIVEQ so that they may continue execution. DELETE is called to indicate that the path is no longer eligible for evaluation. TERMV uses LOCKP to insure that no path attempts to WAITV for the path's value while it is in the process of terminating the path.

A CREATED path may return a value implicitly by exiting the outermost procedure call in its environment, in which case the value returned is the result returned by the procedure. When control underflows in this way, the TERMINATION\FORM 'TERMV(LASTVALUE, MYPATH)' is evaluated which produces the desired effect, c.f. 2.2.6.

```

    TERMV <- (V:ANY, P:ARPTR; NONE)
    BEGIN
        DECL Q:ARPTR;
        DECL B:BOOL;
        B <- LOCKP(P)
        P # MYPATH -> SUSPEND(P);
        P.PVALRET => TERM\ERROR();
        V # NIL ->
        P.PAVAL <-
        BEGIN

```

```

        MD(V).CLASS="PTR" => V;
        ALLOC(MD(V) LIKE V)
    END;
    P.PVALRET <- TRUE;
    Q <- P.PVALQ.FIRST;
    *
    WHILE Q # NIL DO
        BEGIN
            REMOVE(Q, P.PVALQ);
            ENTER\INACTIVEQ(Q);
            Q<-P.PVALQ.FIRST
        END;
    P#MYPATH => [ ]DELETE(P); B => UNLOCKP(P) [ ];
    DELETE(P)
    NT DELETE is described below;
END;

```

The procedure DELETE may be used to indicate that a concurrent path is no longer eligible for evaluation. DELETE suspends the path if necessary, and then calls DELETE\PATH. Self-deletion always leaves the PAL unlocked so that other paths may access the deleted path's ACTRC.

```

DELETE <- EXPR(P:ARPTR; NONE)
BEGIN
    DECL B:BOOL;
    MYPATH#PCIAR =>
        BEGIN
            B <- LOCKP(P);
            P#MYPATH -> SUSPEND(P);
            CIA("DELETE",P);
            B -> UNLOCK(P)
        END;
    DELETE\PATH(P);
    P=LASTRUN =>
        BEGIN
            P.DORMANT <- TRUE;
            UNLOCKP(P);
            LASTRUN <- NIL
        END
END;

```

Since the procedure CREATE uses the control primitive PAP to initialize the computation to be performed by the

*
 'WHILE f DO g' is an abbreviation for an iteration with a zero step, i.e. it is equivalent to 'FOR I<-1,1, ..., N WHILE f DO g'.

concurrent path, there are no restrictions placed upon the evaluation of paths so CREATED. In particular, there are no constraints placed upon the intra-path control of an individual path. Such freedom is feasible since no path can directly reference the environment of another. All shared data lies in the heap.

A fork statement, c.f. 1.2.1, can be used to produce a multi-path organization in which one control path creates a set of paths to be evaluated concurrently and the creator path resumes execution only after all FORKed control paths have completed their execution. In this restricted control regime the FORKed paths may obtain references to the environment of their creator since the creator is constrained to wait for their termination. The procedure FORK can be used to effect this organization. FORK takes a list of procedure calls as its single argument. For each element of the list it allocates a dependent path, c.f. 2.2.8, and DPAPs the procedure call into the path. The new paths are put on the INACTIVEQ so that they will be evaluated concurrently and then FORK waits for all of the paths to terminate. FORK returns a ROW(REF) whose components are the values returned (in the sense of WAITV and TERMV) by the FORKed paths.

```

FORK <- EXPR(FORKL:FORM LISTED; ROW(REF))
  BEGIN
    DECL FORKN:INT BYVAL LENGTHL(FORKL);
    DECL FORKF:ROW(ARPTR) BYVAL
      CONST(ROW(ARPTR) SIZE FORKN);
    DECL FORKV:ROW(REF) BYVAL

```



```

        CONST(ROW(REF) SIZE FORKN);
FOR FORKI<-1, ..., FORKN DO
    BEGIN
        FORKP[FORKI]<-DPAP(FORKL.CAR,
                           MDEP(GET\PATH(1)));
        FORKP[FORKI].TERMINATION\FORM<-
            QUOTE(TERMV(LAST\VALUE,MYPATH));
        ENTER\INACTIVEQ(FORKP[FORKI]);
        NT Start the path;
        FORKL<-FORKL.CDR;
        NT Next procedure call
        FORKL#NIL => NOTHING;
        FOR I<-1 ,..., FORKN DO
            FORKV[I]<-WAITV(FORKP[I])
        NT Wait for all paths to terminate;
    END;
FORKV
NT Return the ROW of values;
END;

LENGTHL <- FXPR(F:FORM;INT)
BEGIN
    F = NIL => 0;
    1 + LENGTHL(F.CDR)
END;

```

The advantage of the FORK organization is that it allows an argument to a concurrent process to be passed BYREF, even if the argument is a stack object. Hence, a path can construct a 'large object' on its stack and then pass it to concurrent paths without causing the object to be copied. With CREATE it would be necessary to allocate the object in the heap in order to allow concurrent paths to access it. For example, suppose we would like to compute

$$C \leftarrow (A \text{ MM } B) \text{ MA } (B \text{ MM } A);$$

where A, B and C are NxN matrices and MM and MA represent matrix multiplication and addition, respectively. Let us also assume that we would like to perform the two matrix multiplications concurrently and to compute the sum of the

two intermediate matrices as soon as each row is available. We will create 3 FORKs. The first fork will compute $B \text{ MM } A$, the second will compute $A \text{ MM } B$, and the third will compute the sum. The first two forks will signal the third, using semaphores, every time they have completed a row. Assuming that $M \leftarrow \text{ROW}(N, \text{ROW}(N, \text{INT}))$ and that N , A , B and C are defined in the environment, we have

```
BEGIN
  DECL LA, AB:M;
  DECL SAB, SBA:SEM BYVAL ALLOC(SEM\ELT);
  FORK(MUL(A, B, AB, SAB),
        MUL(B, A, BA, SBA),
        SUM(C, AB, BA, SAB, SBA))
END;
```

where

```
MUL <- FXPR(X:M, Y:M, Z:M, S:SEM; NONE)
BEGIN
  FOR I<-1,...,N DO
    FOR J<-1,...,N DO
      BEGIN
        Z[I][J] <-
          BEGIN
            FOR L<-1,...,N DO
              S<-S + X[I][L] * Y[L][J];
            S
          END;
        V(S)
        NT Indicate row completed;
      END
    END;
  END;
```

and

```
SUM <- EXPR(RES:M, X:M, Y:M, SX:SEM, SY:SEM; NONE)
BEGIN
  FOR I <- 1,...,N DO
    BEGIN
      P(SX);
      NT Wait for row of X;
      P(SY);
      NT Wait for row of Y;
      NT Row I is ready;
      FOR J <- 1,...,N DO
        RES[I][J] <- X[I][J]+Y[I][J]
```

END
END;

Since FORK is used, the matrices A, B, C and AB are not copied when passed as arguments to the concurrent paths.

4. SIMULATION

In this section, we will present a set of MPEL1 procedures which may be used to effect a clock driven simulation. We will use the organization and terminology of the simulation language SIMUA [Da66]. Hence, this section also demonstrates how the control structure of an existing language may be synthesized using the primitives and framework of MPEL1.

In SIMULA, c.f 1.2.1, a simulation consists of the processing of a time ordered sequence of events (called event notices.) Associated with each event notice is the system time at which it is to occur and a single process whose evaluation constitutes the 'processing' of the event. Processes may delete event notices, thereby canceling the event, and schedule new events by including an event notice (with an associated process) in the sequence of event notices. Although many events may be set to occur at the same system time, the associated processes are evaluated sequentially. Hence, only one process is active at any

given time.

Before we can be more precise, we must introduce some data definitions.

```

SEPTR <- PTR(SET\DESC, ELT\DESC);
SET\DESC <- STRUCT(SUC:SEPTR, PRED:SEPTR);
ELE\DESC <- STRUCT(SUC:SEPTR,
                  PRED:SEPTR,
                  PROCESS:ARPTR);

SET <- PTR(SET\DESC);
ELEMENT <- PTR(ELE\DESC);
EVENTN <- PTR(EVNT\DESC);
EVNT\DESC <- STRUCT(EVTIME:INT,
                  NEXTEV:EVENTN,
                  PREVEV:EVENTN,
                  ELM:ELEMENT);

ACTRC <- (... , EVN:EVENTN, ...);
NT EVN is an extended component;

```

A SET describes an ordered sequence of set ELEMENTS. There is one permanent member of the set, namely, the SET\DESC. If S is a SET, then S is empty if and only if S.PRED=S.SUC=S. Otherwise, S and the ELEMENTs of the set form a doubly-linked circular list. Each ELEMENT contains a PROCESS field which points to the ACTRC of a path. Hence, we may say that a SET describes a set of processes. Since the association is indirect (through an ELT\DESC) a process may be a member of more than one set. An ELEMENT, however, may be a member of only one set at a time.

We define four procedures which operate on sets. FIRST and LAST return the first and last elements of a set, respectively, or NIL if the set is empty. INCLUDE adds an ELEMENT to a SET (at the 'end') and EXTRACT removes an ELEMENT from a SET.

```

FIRST <- EXPR(S:SET; ELEMENT)
  BEGIN
    S.SUC=S => NIL;
    S.SUC
  END;

LAST <- EXPR(S:SET, ELEMENT)
  BEGIN
    S.PRED=S => NIL;
    S.PRED
  END

INCLUDE <- EXPR(E:ELEMENT, S:SET; NONE)
  BEGIN
    E.PRED <- S.PRED;
    E.SUC <- S;
    S.PRED.SUC <- E;
    S.PRED <- E
  END

EXTRACT <- EXPR(E:ELEMENT; NONE)
  BEGIN
    E.PRED=E.SUC=NIL => NOTHING;
    E.PRED.SUC <- E.SUC;
    E.SUC.PRED <- E.PRED
  END

```

An event-notice (EVENTN) is a pointer to an object of mode EVNT\DESC. Associated with each EVENTN is the system time at which it is to occur (EVTIME), pointers to the next and previous event notices or NIL if the event notice is the last or current one, and an ELEMENT whose PROCESS field gives the process (path) to be evaluated. The global

*

To shorten the discussion, we will not distinguish between pointers and the objects to which they point.

variable CURRENT, which is of mode EVENTN, references the doubly-linked list of event notices which is ordered according to non-decreasing values of the EVTIME components. This list, called the sequencing set (SQS), describes the events which constitute the simulation. If a process is not referenced from the SQS, then it is said to be passive and its EVN component is NIL. Otherwise, the EVN component points to the event notice which references the process, i.e. if C is an EVENTN in the SQS then

$$C.ELE.PROCESS.EVN = C$$

Hence, a process may be associated with at most one event notice.

The process currently being evaluated is the one referenced by CURRENT.ELE.PROCESS. The current system time is CURRENT.EVTIME. The NOFIX operators CUR and TIME return the ELEMENT and the system time associated with the CURRENT event notice, respectively. The time reference of a EVENTN may be obtained using the procedure EVTIME.

```
CUR <- EXPR(;ELEMENT) CURRENT.ELM;
```

```
EVTIME <- EXPR(E:ELEMENT; INT)
```

```
  BEGIN
```

```
    E.PROCESS.EVN=NIL => 0;
```

```
    E.PROCESS.EVN.EVTIME
```

```
  END;
```

```
TIME <- EXPR(;INT) (EVTIME(CUR));
```

We may now describe the procedures which operate on the SQS, and thus provide the means whereby processes can affect the scheduling of events. In all cases, the procedures take

ELEMENTs as arguments. Hence, references to event notices or processes are always indirect.

The procedure CANCEL removes the event notice associated with the referenced processes (if one exists) from the SQS, thereby canceling the event and making the process passive. TERMINATE has the same effect as CANCEL, except that the process may never be reactivated, i.e. it is ineligible for evaluation in the sense of DELETE\PATH. CANCEL(CUR) will cause control to be transferred to the next process on the SQS; the associated event notice becomes CURRENT. In this case, CANCEL uses the control primitive CIA to transfer control to the appropriate process.

```

CANCEL <- EXPR(I:ELEMENT; NONE)
  BEGIN
    E.PROCESS.EVN=NIL => NOTHING;
    DELEVN(E.PROCESS.EVN) => CIA("PASS",CURRENT)
  END

PASS <- EXPR(EV:EVENTN; NONE)
  BEGIN
    LASTRUN <- EV.ELM.PROCESS;
    LASTRUN=NIL => SIMERROR()
  END;

DELEVN <- EXPR(EV:EVENTN; BOOL)
  BEGIN
    EV.PROCESS.EVN <- NIL;
    NT Path is passive;
    EV#CURRENT => [ ]EV.PREVEV.NEXTEV<-EV.NEXTEV;
    EV.NEXTEV=NIL=> FALSE;
    EV.SUC.PREVEV<-EV.PREVEV;
    FALSE[ ];
    CURRENT <- CURRENT.NEXTEV;
    NT CURRENT is deleted;
    CURRENT.PREV<-NIL;
    TRUE
    NT Return TRUE if the current EVENTN
      has been deleted;
  END;

```

```

TERMINATE <- EXPR(E:ELEMENT; NONE)
BEGIN
  E.PROCESS.EVN=NIL => NOTHING;
  DFLEVN(E.PROCESS.EVN) => CIA("TERMPASS", CURRENT);
  CIA("DELETE\PATH",E.PROCESS)
END;

TERMPASS <- EXPR(EV:EVENT; NONE)
BEGIN
  DELETF\PATH(LASTRUN);
  NT Path may not be reactivated;
  PASS(EV)
END;

```

The procedure NEWPROC takes a procedure call to be evaluated in the environment of a new process (in the sense of PAP.) NEWPROC creates a new ELEMENT and a new path and uses PAP to initialize the environment of the path. The TERMINATION\FORM is set to be TERMINATE(CUR). Hence, exit from the PAPed procedure will cause the process to be TERMINATED. NEWPROC returns the ELEMENT which references the new passive process.

```

NEWPROC <- EXPR(F:FORM UNEVAL; ELEMENT)
BEGIN
  DECL F:ELEMENT BYVAL
  ALLOC(ELE\DESC OF NIL,NIL,GET\PATH(1));
  E.PROCESS.TERMINATION\FORM <-
    QUOTE(TERMINATE(CUR));
  PAP(F,E.PROCESS);
  E
END;

```

The two procedures, ACTIVATE and REACTIVATE may be used to schedule future events. In both cases, the first argument E (an ELEMENT) specifies the process to be associated with the new event. In the former case, the process must be passive. Otherwise, no scheduling takes place. In the latter case, if the process is not passive,

then the associated event notice is deleted and the event is essentially re-scheduled. The other arguments (T,AFTER,E2) to both procedures determine where in the SQS the new event notice, say N, is to be inserted. If E2 is non-null and if E2.PROCESS is not passive, then N is inserted before or after E2.ELE.PROCESS.EVN as AFTER is FALSE or TRUE, respectively. Otherwise, if E2 is null, then N is inserted before or after all event notices at time T ($T=0 \Rightarrow \text{TIME}$) as AFTER is FALSE or TRUE. Hence, if E is an element whose process is passive, then all of the following transfer control to the process referenced by E.

ACTIVATE (E)

ACTIVATE(E, TIME)

ACTIVATE(E,0,FALSE,CUR)

```

REACTIVATE <-
  EXPR(E:ELEMENT, T:INT, AFTER:BOOL, E2:ELEMENT; NONE)
  BEGIN
    E2#NIL AND E2.PROCESS.EVN=NIL => NOTHING;
    E.PROCESS.EVN#NIL => DELEVN(E.PROCESS.EVN);
    SCHEDULE(E, [ ) E2#NIL => EVTIME(E2);
    T=0 => TIME;
    T( ], AFTER, E2)
  END;

```

```

ACTIVATE <-
  EXPR(E:ELEMENT, T:INT, AFTER:BOOL, E2:ELEMENT; NONE)
BEGIN
  E2#NIL AND E2.PROCESS.EVN=NIL => NOTHING;
  E.PROCESS.EVN#NIL => NOTHING;
  SCHEDULE(E, [ )E2#NIL => EVTIME(E2);
                    T=0 => TIME;
                    T(], AFTER, E2)
END;

```



```

SCHEDULE <-
  EXPR(E:ELEMENT, T:INT, AFTER:BOOL, E2:ELEMENT; NONE)
  BEGIN
    DECL TE: INT BYVAL TIME;
    DECL SC, C:EVENTN BYVAL CURRENT;
    DECL N:EVENTN BYVAL
      ALLOC(EVENT\DESC OF T,NIL,NIL,E);
    BEGIN
      E2 # NIL =>
        INSERT(N, E2.PROCESS.EVN, AFTER);
      T LT TIME => SIM\ERROR();
    L: C.NEXTEV = NIL => INSERT(N, C, AFTER);
      AND(C.EVTIME LT T,
        C.NEXTEV.EVTIME GE T,
        NOT AFTER)
      OR
      AND(C.EVTIME LE T,
        C.NEXTEV.EVTIME GE T,
        AFTER) =>
        INSERT(N, C.NEXTEV, FALSE);
      C <- C.NEXTEV;
      GOTO L
    END;
    N.FLM.PROCESS.EVN <- N;
    NT Process is not passive;
    CURRENT # SC => CIA("PASS", CURRENT);
  END;

```

```

INSERT <-
  EXPR(NEW:EVENTN, OLD:EVENTN, AFTER:BOOL; NONE)
  BEGIN
    AFTER =>
      BEGIN
        NEW.NEXTEV <- OLD.NEXTEV;
        NEW.PREVEV <- OLD;
        OLD.NEXTEV <- NEW;
        NEW.NEXTEV # NIL =>
          NEW.NEXTEV.PREVEV <- NEW;
      END;
    NEW.NEXTEV <- OLD;
    NEW.PREVEV <- OLD.PREVEV;
    OLD.PREVEV <- NEW;
    NEW.PREVEV # NIL => NEW.PREV.NEXTEV <- NEW;
    OLD = CURRENT => CURRENT <- NEW
  END;

```

The procedure HOLD makes the current process be inactive for X units of system time.

```
HOLD <- EXPR(X:INT;NONE) (REACTIVATE(CUR, TIME+X, TRUE));
```

To illustrate the use of the procedures described above, we will present a set of MPOL1 procedures which describe a simple epidemic model (as defined in [Da66].)

A contagious, nonlethal disease is circulating through a fixed size POPULATION. To combat the disease, certain actions are taken by a public health organization. When an individual is infected, he is noncontagious for INCUBATION days (during which he has no SYMPTOMS,) after which he is contagious for LENGTHI days (during which he has SYMPTOMS.) Each DAY of the latter period he may seek TREATMENT, in which case he is immediately and permanently cured, (i.e he becomes immune.) The probability of seeking treatment on any given day I is PROBTREAT[I]. Each contagious sick person has some number of CONTACTs per day. For each contact, the probability of infecting an uninfected person is PRINF. An untreated infection terminates after LENGTHI days.

Two types of processes are created - one to represent a SICK\PERSON and one to represent the TREATMENT of an individual. In the former case, each SICK\PERSON has an environment (ENV) which is the set of people he has infected. TREATMENT processes represent the public countermeasures taken against the disease. A patient is removed from his environment. If he has SYMPTOMS then he is immediately cured; his ENV is searched and each member is treated. Otherwise, the patient is given a 'cheap pill'

which has a probability PROBMASS of success. His ENV is not searched. The simulation ends after SIMPERIOD days.

POPULATION, LENGTHI, CONTACTS, SIMPERIOD, INCUBATION, PRINF, PROBMASS, and PROBTREAT are assumed as global constants. In addition, U1, U2, U3 and U4 are NOFIX operators which represent different streams of pseudo-random numbers. The procedure call DRAW(PROB,U1) makes a random drawing with probability PROB of success, in which case it returns TRUE. The procedure call POISSON(CONTACTS,U2) returns a random drawing from the Poisson distribution with mean CONTACTS. UNINFECTED is used to record the number of UNINFECTED people in the population.

The simulation begins with one path that executes the block presented below. The SQS is initialized to contain a single event-notice for the current path. The first SICK\PERSON process is activated, after which the initial path holds for the duration of the simulation.

```
BEGIN
  CURRENT <- ALLOC(EVNT\DESC OF
    O,NIL,NIL,ALLOC(ELEM\DESC OF NIL,NIL,MYPATH));
  CURRENT.ELM.PROCESS.EVN <- CURRENT;
  NT Not passive;
  UNINFECTED <- POPULATION;
  ACTIVATE(NEWPROC(SICK\PERSON()));
  HOLD(SIMPERIOD)
END;
```

The procedures SICK\PERSON, INFECT, and TREATMENT are defined as follows.


```

SICK\PERSON <- EXPR(; NONE)
BEGIN
  DECL SYMPTOMS:BOOL;
  DECL ENV:SET BYVAL ALLOC(SET\DESC);
  UNINFECTED <- UNINFECTED - 1;
  SYMPTOMS <- FALSE;
  HOLD(INCUBATION);
  NT Wait till the end of the
    incubation period;
  SYMPTOMS <- TRUE;
  FOR DAY <- 1, ... , LENGTHI DO
    BEGIN
      NT Either seek treatment or infect
        some contacts;
      DRAW(PROBTEAT[DAY],U1) =>
        ACTIVATE(NEWPROC(TREATMENT(CUR)));
      INFECT(POISSON(CONTACTS,U2),ENV);
      HOLD(1)
      NT Wait one day;
    END
  END;
END;

INFECT <-
  EXPR(N:INT, S:INT; NONE)
  BEGIN
    NT N is the number of contacts;
    NT S is the ENV;
    FOR I <- 1, ... , N DO
      BEGIN
        DRAW(PRINF * UNINFECTED/POPULATION,U3) =>
          BEGIN
            INCLUDE(NEWPROC(SICK\PERSON()),S);
            NT Infect one person;
            ACTIVATE(LAST(S))
            NT Start him now;
          END
      END
    END
  END;
END;

```

```

TREATMENT <-
  EXPR(PATIENT:ELEMENT; NONE)
  BEGIN
    DECL ENV:SET;
    DECL SYMPTOMS:BOOL;
    EXTRACT(PATIENT);
    NT Remove patient from ENV;
    ENV <- PFETCH("ENV",PATIENT.PROCESS);
    SYMPTOMS <- PFETCH("SYMPTOMS",PATIENT.PROCESS);
    NT PFETCH the values of SYMPTOMS
      and ENV from the SICK\PERSON being treated;
    SYMPTOMS =>
      BEGIN
        TERMINATE(PATIENT);
        WHILE FIRST(ENV) # NIL DO
          NT Treat each person in the ENV;
          (ACTIVATE(NEWPROC(TREATMENT(FIRST(ENV))))
          NT FIRST(ENV) is deleted from the set
            ENV upon activation of the process;
        END;
        DEAW(PROBMASS,U4) => TERMINATE(PATIENT)
      NT Otherwise, leave the patient in the system,
        but removed from his ENV;
    END;
  END;

```

5. MONITORING AND RELATIVE CONTINUITY

Control primitives which allow a variable to be monitored for changes in its value have appeared in a number of languages. For example, the WAITUNTIL primitive of PPL [Po72] and Fisher's monitor primitive [Fi70] allow a process to be resumed as soon as an associated condition becomes TRUE, c.f. 1.2.1, 1.2.3. Here, we discuss how monitoring can be realized in MPEL1.

Let us assume that we desire a function of the form

MONITOR(X,V,EXP)

*

where X is a variable of mode integer, V is an integer and EXP is a FORM. The interpretation of MONITOR is that when $X=V$, then EXP is to be evaluated. If $X=V$ when the MONITOR is executed, then EXP is evaluated immediately. It should be clear that it is sufficient to monitor assignments to X, i.e. if it is possible to obtain control every time the variable X is assigned a new value, then we can surely detect when it is given the value V. This ability is provided by the extended mode facility of FL1, c.f. 5.1.2.

In particular, the extended mode facility allows the user to define a new mode, say SINT (sensitive-integer), which is identical to the mode INT except that a

*

We will restrict our discussion to integers. Monitoring of other data types can be achieved by analogous techniques.

user-defined assignment function will be called whenever an object of mode SINT is to be assigned. For example, we can define SINT, MONITOR and the SINT assignment function as follows.

```
SINT <- STRUCT(I:INT, V:INT, EXP:FORM, MFLG:BOOL);
```

```
MONITOR <-
  EXPR(X:SINT, V:INT, F:EXP; NONE)
  BEGIN
    NOT UR(X).MFLG =>
      BEGIN
        UR(X).I = V => EVAL(F);
        UR(X).V <- V;
        UR(X).EXP <- F;
        MFLG <- TRUE
      END;
    MONITOR\ERROR()
  END;
```

```
SINT\ASSIGN <-
  EXPR(S:SINT, V:INT; INT)
  BEGIN
    UR(S).I <- V; NT V is the new value;
    NOT UR(S).MFLG => V;
    UR(S).V = V =>
      BEGIN
        UR(S).MFLG <- FALSE;
        EVAL(UR(S).EXP);
        V
      END;
    V
  END;
```

If X is a SINT, $UR(X).I$ is the actual integer, $UR(X).MFLG$ is TRUE if and only if the integer is being monitored, $UR(X).V$ is the value being monitored for, and $UR(X).EXP$ is the expression to be evaluated. Whenever an

*

The procedure UR is used to indicate that selection is to be performed upon the underlying representation of SINT. This is required since it is also possible to construct user defined selection functions.

assignment to a SINT is made, the procedure `SINT\ASSIGN` is called. If the particular SINT is being monitored and if it will now be equal to $UR(S).V$, then `EXP` is evaluated and the monitor is turned off.*

Two points should be stressed. First, SINTs act just like INTs except for assignment. Whenever a SINT is in hand and an INT is required, the SINT is treated as if it were an integer, using a user-defined conversion function [Weg70][Weg71]. Second, only SINT assignments are affected; the overheads associated with monitoring are not passed on to all INTs.

In the example above, `EXP` will be evaluated in the environment of whichever path performs the assignment that sets the SINT to the specified value. Other semantics for `MONITOR` operations are possible. For example, Fisher's monitor operation constructs a new process in which `EXP` is to be evaluated. The value returned by monitor is a reference to this process (which is also apparently cyclically testing the condition.) The operation unmonitor may be used to destroy a monitor process. It is possible to construct many monitor processes which are all testing the same variable.

*

Here, for simplicity, we have limited ourselves to one `EXP` per SINT. The general case is discussed below.

A description of monitor would be a straightforward extension of the example above, were it not for the subtle relation between monitor and another of Fisher's primitives, namely, the cont operation. We will first discuss how the cont operation may be realized in MPEL1 and then describe a monitor operation which is consistent with this realization and with Fisher's definition.

If a process executes cont(exp) then the expression will be evaluated relatively continuous with respect to the evaluation of other processes; all other processes must pause while the exp is evaluated. If exp creates new processes, then they inherit the level of relative continuity of their creator. Hence, many levels of relative continuity can be created. At any given time, only those processes at the highest level may be evaluated. When higher-level processes terminate, processes at lower levels are allowed to continue execution.

In MPEL1, we will replace cont by the two NOFIX operators STARTRC and ENDRC. STARTRC indicates that the level of relative continuity of the path should be increased by one. ENDRC indicates that the level should be decreased by one. Thus, in MPEL1, cont(exp) is replaced by

```
[ ] STARTRC; exp; ENDRC [ ]
```

The implementation of relative continuity is straightforward. We include the integer component LEVEL as

an extended component of the mode ACTRC. LEVEL specifies the level of relative continuity at which the path is evaluating. Initially, all paths have LEVEL=0. The path-scheduler is redefined, using the techniques of section 2.4, to include the integer RCLEVEL in the CI environment and to maintain paths on the INACTIVEQ^{*} in the order of their level component (highest to lowest.) RCLEVEL indicates the level of relative continuity at which the system is being evaluated. The procedure CREATE is modified to give the path created the same level of relative continuity as the creator.

When a path P executes STARTRC, RCLEVEL and P.LEVEL are incremented by one. If there exist active paths at lower levels of relative continuity, then each of these paths is interrupted (using STOP\PATH) and forced to pass control to the CI and execute RELSTOP. When each path passes control to the CI, it is put on the INACTIVEQ. When the last of these is processed, then P is allowed to continue execution. The mechanism used here is similar to the one employed by SUSPEND, c.f. 3.3.

*

We also assume that the procedures of section 3.3 are redefined to use the procedure INSERTL (as opposed to ENTERL) to place paths on the INACTIVEQ. INSERTL places the path on the queue at the appropriate point according to its LEVEL component.

```

STARTRC <- EXPR(; NONE)
BEGIN
  MYPATH#PCIAR => CIA("STARTRC");
  LASTRUN.LEVEL <- RCLEVEL <- RCLEVEL+1;
  UPRC()
END;

UPRC <- EXPR(; NONE)
BEGIN
  DFCL N:INT BYVAL LASTRUN.LEVEL;
  FOR I <- 1, ..., NPROC DO
    BEGIN
      PAVECT[I].CURPATH # PAVECT[I].IDLEPATH
      AND
      PAVECT[I].CURPATH.LEVEL LT N =>
      BEGIN
        PIVECT[I]<-
          CONS(LIST("RELSTOP",
                    ALLOC(REF LIKE
                          ALLOC(ARPTR LIKE
                                LASTRUN))),
              PIVECT[PROC]);
        STOP\FATH(PAVECT[I].CURPATH);
        B <- TRUE
      END;
    END;
    B => LASTRUN <- NIL
  END;

RELSTOP <- EXPR(P:ARPTR; NONE)
BEGIN
  DECL B:BOOL;
  INSERTL(LASTRUN);
  LASTRUN <- NIL;
  FOR I <- 1, ..., N TILL B DO
    BEGIN
      AND(PAVECT[I].CURPATH#PAVECT[I].IDLEPTH,
          I#PROCNUM,
          PAVECT[I].CURPATH.LEVEL LT RCLEVEL)
      => B <- TRUE
    END;
    B => NOTHING;
    NT Not all paths at lower levels
    have stopped;
    INSERTL(P) NT All have stopped;
  END;

```

When a path P executes ENDRC, then P.LEVEL is decremented by one and the path is inserted at the appropriate point in the INACTIVEQ. If there still exists

paths (either active or on the INACTIVEQ)^{*} at the current RCLEVEL, then no further action occurs. Otherwise, RCLEVEL is set to be equal to the LEVEL of the first path on the INACTIVEQ.

```

ENDRC <- EXPR( ; NONE)
  BEGIN
    DECL B:BOOL
    MYPATH#PCIR => CIA("ENDRC");
    LASTRUN.LEVEL <- LASTRUN.LEVEL-1;
    INSERTL(LASTRUN);
    LASTRUN <- NIL;
    FOR I <- 1,..., NPROC TILL B DO
      BEGIN
        I=PROCNUM => NOTHING
        PAVECT[I].CURPATH#PAVECT[I].IDLEPATH
          AND
        PAVECT[I].CURPATH.LEVEL=RCLEVEL
          => B <- TRUE
      END;
    B => NOTHING;
    NT There exists an active path
      at the current RCLEVEL;
    RCLEVEL GT INACTIVEQ.FIRST LEVEL =>
      RCLEVEL<-INACTIVEQ.FIRST.LEVEL
  END;

```

RCSCHEDULE is used to perform path scheduling. Let P be the first non-DORMANT path on the INACTIVEQ. If P.LEVEL is less than RCLEVEL, then no path is scheduled. If P.LEVEL is equal to RCLEVEL, then P is scheduled. Otherwise, if P.LEVEL is greater than RCLEVEL, then RCLEVEL is set to P.LEVEL, LASTRUN is set to P and UPRC is called to determine if any active paths are at a lower level of relative continuity, and if so, to interrupt them. The latter case

*

If a path is not in the running set (e.g. it has performed a P operation and is on the semaphore's WLIST,) then its level is not counted until it again becomes a running path. See the discussion of monitor paths below.

can occur when a monitor process is activated as described below.

```

RCSCHEDULE <- EXPR(; ARPTR)
BEGIN
  DECL Y:ARPTR;
  Y <- INACTIVEQ.FIRST;
L: Y=NIL => NIL;
  RCLEVEL GT Y.LEVEL => NIL;
  RCLEVEL=Y.FIRST.LEVEL AND NOT Y.DORMANT =>
    BEGIN
      REMOVE(Y,INACTIVEQ);
      Y
    END;
  RCLEVEL LT Y.FIRST.LEVEL =>
    BEGIN
      LASTRUN <- Y.FIRST;
      REMOVE(LASTRUN,INACTIVEQ);
      RCLEVEL <- LASTRUN.LEVEL;
      UPRC();
      LASTRUN
    END;
  Y <- Y.NEXT;
  GOTO L
END;

```

INSERTL is used to place paths on the INACTIVEQ. If the LEVEL of the path P to be inserted is greater than the current RCLEVEL, then LASTRUN (the current path) is inserted, LASTRUN is set to NIL, and then P is inserted. Hence, scheduling is forced. Since P will be first on the INACTIVEQ, and RCLEVEL is less than P.LEVEL, then RCSCHEDULE will interrupt all active paths at lower LEVELs and then allow P to be evaluated. If P.LEVEL is less than or equal to RCLEVEL, then P is simply placed at the appropriate point in the queue.

```

INSERTL <- EXPR(P:ARPTR); NONE)
BEGIN
  DECL Q:ARQPTR BYREF INACTIVEQ;
  DECL Y:ARPTR BYVAL Q.FIRST;
  LASTRUN#NIL AND P.LEVEL GT RCLEVEL ->
    BEGIN

```

```

        INSERTL(LASTRUN);
        LASTRUN <- NIL
        NT Force a call to RCSCHEDULE;
    END;
    Y=NIL => ENTERL(P,Q);
    Y.LEVEL LE P.LEVEL =>
    BEGIN
        Q.FIRST <- P;
        P.NEXT <- Y
    END;
    Q.LAST GE P.LEVEL =>
    BEGIN
        P.NEXT <- NIL;
        Q.LAST.NEXT <- P;
        Q.LAST <- P
    END;
    Y.NEXT=NIL => ENTERL(P,Q);
    Y.NEXT.LEVEL LE P.LEVEL =>
    BEGIN
        P.NEXT <- Y.NEXT;
        Y.NEXT <- P
    END
END;

```

We can now describe the complete semantics of the monitor operation. Here, SINT is defined as follows:

```

SINT <- PTR(STRUCT(INTP:INT, MLIST:MPTR));
MPTR <- PTR(MELT);
MELT <- STRUCT(VALUE:INT, PATH:ARPTR, NEXT:MPTR);

```

The MLIST is a list of the processes monitoring the SINT. Each MELT specifies the VALUE to be checked and the path to be evaluated.

The procedure MONITOR creates a path P (which is added to the MLIST of N) to be activated when N is assigned the value V. F is the form to be evaluated by P. P is given the same level of relative continuity as the path executing the MONITOR. In addition, P is enabled for the interrupt "UNMON" to allow for unmonitoring. It is important to note

that since P is not considered to be a running path (it is neither active nor on the INACTIVEQ), P.LEVEL has no effect upon RCLEVEL. Hence, when P is activated RCLEVEL may be higher or lower than P.LEVEL. In the former case, P will not be evaluated until RCLEVEL is lowered to P.LEVEL. In the latter case, P has priority over all other paths. Hence, they must be interrupted so that P may evaluate. INSERTL detects this fact and forces scheduling to achieve the desired effect. If P=RCLEVEL, then the interpretation is straightforward.

The definitions of MONITOR and SINT\ASSIGN are as follows.

```

MONITOR <- EXPR(N:SINT,V:INT, F:FORM UNEVAL; ARPTR)
  BEGIN
    DECL P:ARPTR BYVAL PAQ(EVAL(F), GET\PATH(1));
    DECL M:MPTR BYVAL ALLOC(MELT OF V,P,NIL);
    P.LEVEL <- MYPATH.LEVEL;
    NT Relative continuity level is inherited;
    ENABLE\PATH("UNMON",3,QUOTE(DELETE(MYPATH)),P);
    NT To allow for unmonitoring;
    *
    CIA("MONITOR1",M,N,V) ;
    P
  END;

MONITOR1 <- EXPR(M:MPTR, N:SINT, V:REF; NONE)
  BEGIN
    UR(N).VALUE = VAL(V) =>
      BEGIN
        INSERTL(M.PATH, INACTIVEQ);
        RUNSET\FLAG <- TRUE
      END;
    M.NEXT <- UR(N).MLIST.NEXT;
    UR(N).MLIST <- M
  END;

```

*

Here, we pass three arguments to the CIA called procedure. This is not consistent with the definition given in section 2.3.1, but can be achieved by extension, c.f. 5.1.1.


```

SINT\ASSIGN <- (N:SINT, Y:INT; INT)
  BEGIN
    CIA("DO\ASSIGN",N,Y);
    Y
  END;

DO\ASSIGN <- EXPR(S:SINT, V:REF; NONE)
  BEGIN
    DECL Y:MLIST BYVAL UR(S).MLIST;
    Y#NIL =>
      WHILE Y.NEXT # NIL DO
        BEGIN
          Y.NEXT.VALUE=VAL(V) =>
            BEGIN
              INSERTL(Y.NEXT.PATH,INACTIVEQ);
              Y.NEXT <- Y.NEXT.NEXT;
            END;
          Y <-Y.NEXT
        END;
      UR(S).MLIST.VALUE=V =>
        BEGIN
          INSERTL(UR(S).MLIST.PATH,INACTIVEQ);
          UR(S).MLIST <- UR(S).MLIST.NEXT;
        END; NT Process first of list;
      NT Now perform assignment;
      UR(S).INTP <- VAL(V)
    END;
  END;

```

For example, consider the following block

```

  BEGIN
    DECL X:SINT BYREF X;
    NT Assume MYPATH.LEVEL=0;
    STARTRC;
    NT MYPATH.LEVEL = 1;
    MONITOR(X,3,QUOTE(PRINT("X=3")));
    NT The LEVEL of the new process is 1;
    X=1 =>
      BEGIN
        STARTRC;
        NT MYPATH.LEVEL =2;
        X <- 3;
        C1:ENDRC;
        NT MYPATH.LEVEL=1;
        C2: ...
      END;
    ENDRC;
    NT MYPATH.LEVEL=0;
    X <- 3;
    C3: ... ;
  END

```

If X initially has the value 1, then the monitor process will not become active until control reaches C2, since RCLEVEL=2 at C1. If, however, X is not initially equal to 1, then the monitor process will become active before control reaches C3, since it is at a higher level of relative continuity than the path performing the assignment. Hence, STARTC, ENTRC and MONITOR effect an interrupt mechanism. If a path creates a monitor process at a higher level of relative continuity, e.g.

```
[ )STARTC; MONITOR(X,Y,f); ENTRC(];
```

then if the condition becomes TRUE, the monitor path will essentially interrupt the original path, since the former is at a higher level. Conversely, the path can mask itself against the effect of the monitor by subsequently executing two STARTRCs to put itself at a higher level than the monitor.

The procedure UNMONITOR, destroys the path P which is monitoring the SINT N. If the path has not been activated, then it is simply deleted. Otherwise, if it is active, it is interrupted (using STOP\PATH) and forced to call DELETE\PATH!. If it is not active, then it is sent an UNMON interrupt (for which all monitor paths are enabled) which will cause the path to delete itself if it ever becomes active.

```

UNMONITOR <- EXPR(P:ARPTR N:SINT; NONE)
BEGIN
  DECL Y:MPTR BYVAL UR(N).MLIST;
  DECL PR:INT;
  MYPATH # PCIAR => CIA("UNMONITOR",P,N);
  BEGIN
    Y.PATH=P => [ ]UR(N).MLIST<-UR(N).MLIST.NEXT;
    DELETE\PATH(P)( );
    WHILE Y.NEXT # NIL DO
      BEGIN
        Y.NEXT.PATH=P => [ ]Y.NEXT<-Y.NEXT.NEXT;
        DELETE\PATH(P)( );
        Y <- Y.NEXT
      END
    END;
    NOT P.ELGFLG => NOTHING;
    NT Otherwise P must be killed;
    FOR I <- 1,..., NPROC TILL PR GT 0 DO
      [ ]PAVECT[I].CURPATH.P => PR<-I( )
    PR=0 => INTERRUPT("UNMON",P);
    STOP\PATH(P);
    PIVECT[I] <-
      CONS(QUOTE(BEGIN
        DELETE\PATH(LASTRUN);
        LASTRUN <- NIL
      END,
        PIVECT[I]));
  END;
END;

```


6. BACKTRACKING

The notion of backtracking [Go65] often allows a more concise representation of an algorithm than would be possible without it. Such an algorithm usually requires that a choice be made at one or more points during its evaluation. If the choice is a 'bad' one, then the algorithm must backtrack to the most recent choice-point at which another choice was available, select a new choice, and then resume execution. Hence, it must be possible to reconstruct (at some later time) the machine state at each choice point. This can be done either explicitly or implicitly. Programs which explicitly handle their own backtracking tend to be obscure and error prone - they must record all changes to data and control. Hence, control procedures which allow for the automatic restoration of machine states are desirable.

Floyd [Fl67] proposes three operations and an implementation technique which allow for automatic backtracking in a flow chart language. The two operations are:

*

A control structure which is related to, but distinct from, backtracking is multi-tracking. In backtracking, choices are processed sequentially - when one value leads to FAILURE, the next value is chosen. With multi-tracking all choices are explored in parallel, via the creation of parallel paths. If a path is unsuccessful, then it notifies its creator who then terminates all other choice paths (and their descendants.)

choice(n) An integer from the set 1, ..., n is chosen.

fail fail 'undoes' all actions performed since the last choice. Another integer, say k, is taken from the choice set and the algorithm continues from the choice-point as if k were the original choice. If all integers from the set have been tried, all actions are undone back to the next previous choice point, etc.

success All accumulated output is printed. If all solutions to a problem are desired, then backtracking is initiated.

Function calls are not permitted in Floyd's language. Hence, to allow reconstruction of machine states it is only necessary to record changes to variables and the control flow through decision points. In a language with function calls, say FL1, it becomes necessary to record the call and block structure (intra-path control) at a choice point so that it may be reinstated upon subsequent failure. Below, we discuss how (single-path) backtracking can be effected using the MPEL1 multi-path facility.

We will assume that the global variable BACKUP references a path which will be used to effect restoration of choice-points. Whenever a new CHOICE is made, control is passed to BACKUP, which then returns control to a COPY of

the original path. Hence, the call structure and the values of variables in the identifier environment are preserved. If an unsuccessful choice has been made then the procedure FAIL can be used to return to the previous choice-point. FAIL simply passes control to the BACKUP path which then returns control to a copy of the original path with a new value from the choice set. SUCCESS takes two arguments. The first specifies a value to be irreversibly printed on an output device. The second is a BOOL which specifies whether or not all solutions are desired. The procedure definitions are as follows.

```

CHOICE <- EXPR(N:INT; INT)
  BEGIN
    PAPO(NEWCHOICE(N,MYPATH),BACKUP);
    CIA("SWITCH\PATHS,BACKUP)
  END;

SWITCH\PATHS <- EXPR(P.ARPTR; NONE)(LASTRUN <- P);

NEWCHOICE <- EXPR(N:INT BYVAL, OLDP:ARPTR; NONE)
  BEGIN
    WHILE N # 0 DO
      BEGIN
        DECL NEWP:ARPTR
        NEWP <- COPY(OLDP);
        PAP(RETFROM("CHOICE",N),NEWP)
        CIA("SWITCH\PATHS", NEWP);
        N <- N-1
      END
    END;

FAIL <- EXPR(; NONE) (CIA("SWITCH\PATHS"),BACKUP));

SUCCESS <- EXPR(SOL: ANY, ALLSOL:BOOL; NONE)
  BEGIN
    PRINT(SOL);
    ALLSOL => FAIL();
    CIA("DELETE\PATH",MYPATH)
  END;

```


Note that if CHOICE is called many times in the program, then nested calls to NEWCHOICE will be Paped into BACKUP. Exit from the procedure NEWCHOICE corresponds to failure of all choices at a given point. Control is thus returned to the call to NEWCHOICE for the previous choice-point.

One additional mechanism is necessary to insure that the complete machine state at a choice-point is restored. The control primitive COPY copies the bindings of variables in the identifier environment of the path. If a variable is bound to a pointer to an object in the heap, then the pointer is copied but the object referenced is not. Hence, when NEWCHOICE returns to a COPY of a path, it is possible that changes to heap objects will not be 'undone.' To insure that heap values are restored correctly, it is necessary to redefine assignment, i.e. "<=". In the procedures below, ASSIGN! is functionally equivalent to the original definition of <= and HEAP is a boolean procedure which returns TRUE if and only if its argument lies in the heap.

```
ASSIGN!(<=, EXPR(VAR:ANY, VAL:ANY; ANY)
  BEGIN
    NOT HEAP(VAR) => ASSIGN!(VAR,VAL);
    PAFO(RESTORE(VAR,VAR),BACKUP);
    ASSIGN!(VAR,VAL)
  END);
```

```
ASSIGN!(RESTORE, EXPR(VAR:ANY; OLDVAL:ANY BYVAL; NONE)
  (ASSIGN!(VAR,OLDVAL)));
```

Whenever an assignment is to be made to a heap object, the procedure RESTORE is Paped into the BACKUP path. The

second argument to RESTORE is passed BYVAL so that the old value may be retained separately. When FAIL transfers control to BACKUP, all heap objects modified since the last choice point will be restored to their original values since the calls to RESTORE are executed in the reverse of the order in which they were PAPed.

Although the procedures described above effect automatic backtracking, the mechanism employed is rather expensive. At each choice point the entire call structure is saved! A similar, but more efficient realization of backtracking is described in [Pr72]. Here, instead of saving the entire machine state at each choice point, only the 'difference' between states is saved. In addition, the programmer may distinguish between assignments which are to be 'undone' upon failure, and those which are not, thus avoiding unnecessary record keeping.

Chapter 4

THE FORMAL DEFINITION OF MPOL

This chapter presents the formal definition of EL1 and the control primitives.

Section 1 discusses some preliminary issues and serves as an introduction to the definition. Sections 2 and 3 present the formal definitions of the constructs of EL1 and the control primitives, respectively. The auxiliary procedures used in the definition are listed in Section 4. Section 5 lists and defines the procedures which are assumed as linguistic primitives. Finally, Section 6 is an index of the modes, variables, procedures, and evaluator labels used in the definition.

1.1 Representation of Programs, Paths and Evaluators

The concrete syntax, or external representation of EL1 is described by the BNF grammar in Appendix 2. An EL1 program is a terminal string derived from the syntactic class <program>. For example,

```
1;  
p(q(a),b,c,);  
[]p=> x<-1 ; 3 (];
```


The abstract syntax, or internal representation of an EL1 program is a list structure which may be defined using the data-type definition facility of EL1. The correspondence between the external and internal representations is specified by augments to the BNF grammar in Appendix 2. Techniques for mapping programs written in concrete syntax into this type of abstract syntax are well known and will not be discussed here. The abstract syntax representations for the programs above are:

```

1
(P (q a) b c)
(BLOCK! (CLAUSE! P (<- x 1) ) 3)

```

The evaluation of an EL1 program is performed in the environment of a path. The environment consists of three related structures: the name-stack, the value-stack, and the control-stack. The name-stack contains an entry for each variable created in the evaluation of the program. Each entry consists of the variable's name and a pointer to its value. The value-stack contains all data objects created by the program which have not been explicitly ALLOCated in the heap. The control-stack describes the current state of the evaluation, i.e. the current nesting of blocks, procedure applications, etc.

The path's activation record contains pointers to the environmental structures described above, fields which are

used to communicate with the control primitives and other path-related information.

A program is evaluated in a path's environment by an evaluator. Only one path may be evaluated by an evaluator at any given time, although it may process different paths at different times. In addition, an evaluator must always be associated with some path, i.e. it must always be kept busy. We will assume that there are some constant number (NPROC) of evaluators available for the simultaneous evaluation of paths.

During its evaluation, an EL1 program may call upon the control primitives to create and delete paths, specify or change the programs to be evaluated in a path's environment, assign evaluators to paths, modify a path's environment, etc.

The formal definition of MP EL1 will consist of a program which defines the I th of NPROC identical evaluators. Since both an MP EL1 program and a path can be represented as EL1 data structures, it is possible to describe an evaluator for the language and the control primitives as a set of MP EL1 procedures. The evaluator described here, however, is written essentially in EL1. The control primitives TSET, CLEAR, EVAL and GOTO are the only ones used by the evaluator. The vast majority of the control primitives are not included in the meta-language.

Thus, the control primitives are explained in terms of (single-path) EL1 and the four primitives listed above, c.f. 5.3.2.

1.2 Evaluator Recursion

The evaluation of an EL1 program is a recursive process, i.e. the evaluator of a given language construct may call upon the evaluators of other constructs, or itself, recursively. For example, nested procedure calls require the procedure application evaluator to be called within itself. Since EL1 procedures are capable of recursion, the number of recursive calls may reach an arbitrary depth. The evaluator however, may not effect the recursion by invoking recursive procedure calls in its own environment, for then if it is subsequently called upon to evaluate another path and the original path is evaluated by another evaluator, the resulting evaluation of both paths will be erroneous. Hence, the evaluators must be 'reentrant' with respect to the paths they are called upon to evaluate. All records relating to the evaluation of a path must be stored in the environment of the path itself.

The control-stack of the path is used in conjunction with a programming discipline to allow an evaluator to obtain the effect of recursion without recursive procedure

calls. There are five issues to be resolved.

- (1) How is the return point specified?
- (2) How are the arguments to the recursive call specified?
- (3) How are the arguments to the current call saved?
- (4) How is the result of a call specified?
- (5) How is the return to the previous call accomplished?

Before we discuss these issues, we must first give a general outline of the structure of the evaluator.

The evaluator is essentially an EL1 BLOCK. The local variables of the block are used to specify the path being evaluated and to hold other temporary values. Corresponding to each construct of the language, (e.g. selection, assignment) there is a labelled sequence of statements which constitute a sub-evaluator for that construct. Corresponding to each control primitive is a labelled sequence of statements which constitute the body of the control subroutine. Control never leaves the block except for calls to procedures used by the evaluator which do not involve recursive evaluation, e.g. to search the name-stack. When an evaluator switches paths, it saves the 'state' of the current path in the corresponding ACTRC, installs the state of the new path, and continues evaluation of the new path from wherever the previous evaluator of the path left off.

To perform a 'recursive' call, the following statements are executed:

```

          PUSHC("TAG")      ;NT Specify return point;
          GOTO FCO           ;NT 'call' FOO;
TAG:      CHECKM(INT)        ;NT Statement to be executed
                               upon return;

```

i.e. the symbolic name of a label to which control is to be transferred upon return is pushed onto the control stack of the path being evaluated. To return, the following statements are executed:

```

          L ← POPC(1)        ;NT L is bound to the return
                               label;
          GOTO EVAL(L)

```

i.e., the return label is popped off the control stack and control is transferred to the labelled statement. To improve the readability of the description of the evaluator, the two sequences of statements above are abbreviated as follows:

```

          CALL FOO;
          CHECKM(INT);

```

and

```

          RETURN;

```

The argument to each sub-evaluator is a pointer to the list structure which is the instance of that construct to be evaluated. The local variable *F* is used to point to the argument. For example, to evaluate

```

V[1] => 3 ;

```

for which the abstract syntax representation is

(CLAUSE! (SEL! V 1) 3)

the clause-evaluator would bind F to the second ^{*}element of the list and call upon the selection-evaluator. The value returned by a sub-evaluator, i.e. the result of evaluating a given construct, is pointed to by the local variable EVRES.

Since any language construct may be invoked recursively, each sub-evaluator must save its argument. To facilitate this, a control mode is associated with each one. When a sub-evaluator is called, an object of the corresponding mode is pushed onto the control stack. The fields of the control mode are used to save the elements of the list structure for this particular call. In addition, the object may contain fields which are used to hold values computed during the evaluation. Thus, the fields of a control mode correspond to the formal parameters and local variables which would be used if the evaluator was able to use recursive procedure calls.

Occasionally, it is useful to allow a control mode to contain a field RETURN which specifies the label to which control is to be returned upon completion of the evaluation. If a return is to be made when the top object on the control

*

A call is actually made to the general evaluator EVAL\FORM which dispatches to the appropriate evaluator, namely EVSEL.

stack is not a SYMBOL, (i.e. the symbolic name of a label,) then the RETURN component of the object is selected and control is passed to the specified point in the evaluator without popping the object off of the stack.

The evaluator described in section 4.2 is not complete because a number of sub-evaluators which were defined in [Weg70] are assumed to be linguistic primitives. In particular, the data type constructors (e.g. ROW, STRUCT) and the object generators (ALLOC and CONST) are not defined. Their omission is justified by the fact that we are primarily concerned with the semantics of EL1 which are directly relevant to the control primitives. In all cases, the omitted sub-evaluators have marginal interaction with the multi-path facility. Hence, their inclusion, although straightforward, would simply lengthen the description of the evaluator and thus weaken this work. The missing sub-evaluators, along with the other linguistic primitives, are defined in section 4.5.

1.3 Stacks

In section 4.1.2, we described a path's environment as consisting of three stacks. Here, we will discuss these stacks in more detail.

Each path possesses a name, value and control stack. They are pointed to by the NS, VS, and CS fields of the path's ACTRC, respectively. Associated with each stack is a current stack index, which is stored in the NP, VP and CP fields of the ACTRC, respectively. The stack index specifies the number of objects which have been 'pushed' onto the stack. When a path is active, the current values for NP, VP and CP are contained in the corresponding evaluator's local variables NP, VP and CP.

The name stack is actually a ROW of STRUCTs, namely

```
NAME\STACK <- ROW(STRUCT(NAME:SYMBOL, VALUE:REF));
```

Hence, entries are pushed onto and popped off of the name stack by storing into the appropriate entry of the row and updating NP appropriately.

The control and value stacks, are objects of mode ^{*}STACK. STACKs have the following properties:

- (1) They may hold objects of any mode.
- (2) They may be indexed as ROWs, e.g. the value of CS[CP] is a pointer to the top object on the control stack, CS[1] is a pointer to the object on the bottom of the stack.
- (3) If an object is 'popped' off of the stack, all

*

STACKs and stack operations as defined here, differ from the definitions given in [Weg70]. We defer justification of the changes until Section 5.3.2.

references to it are set to NIL. Thus it is impossible to retain a pointer to an object which has been removed from a stack.

The following stack operations are defined as linguistic primitives:

(1) `PUSH` \leftarrow `CEXP`(`OBJ:ANY`, `S:STACK;REF`). `OBJ` is copied onto the top of the stack `S`. The number of objects on the stack is increased by one. `PUSH` returns a pointer to the object which has been pushed onto the stack.

(2) `FLUSH` \leftarrow `CEXP`(`S:STACK`, `INDEX:INT; NONE`) If there are `N` objects on the stack `S`, then the `N` th through the `INDEX+1` objects are removed from the stack. Hence, after the `FLUSH`, there remain `INDEX` objects on the stack.

(3) `INSTACK` \leftarrow `CEXP`(`PTR:REF`,
`IND1:INT`,
`IND2:INT, S:STACK; BOOL`)

`INSTACK` returns `TRUE` if and only if the object referenced by `PTR` is on the stack `S` and is one of the objects `S[IND1+1]` through `S[IND2]` or is a sub-object of one of them, i.e. if and only if `PTR` points into the stack between the ranges specified.

(4) `HEAP` \leftarrow `CEXP`(`PTR:REF;BOOL`) returns `TRUE` if and only if `PTR` points to an object which is not on a `STACK`.

Note that `PUSH` and `FLUSH` do not update the stack index

associated with the stack.

A number of stack functions are defined in terms of these primitives to facilitate stack management and the referencing of objects on a stack. For example, TOPC1 is defined as a NOFIX operator and returns a pointer to the top object on the control stack. A complete list of these functions is given in section 4.4.

It is intended that in an implementation of MPEL1 a path's environment will be implemented as three LIFO stacks. Hence, although the mode STACK guarantees that a 'dangling reference' cannot occur (i.e. a reference to an object which has been popped off of the stack), the evaluator must not rely on this property. In particular, since it is possible (via path-dependency) for one path to obtain a reference to the value stack of another, the mechanisms which insure that 'dangling references' do not occur are of central importance in the definition.

1.4 Synchronization

The NPROC evaluators require a means of mutual synchronization. We could, of course, postulate the existence of a CI' which is CIA'-called by the evaluators to effect the synchronization. Unfortunately, this leads to direct circularities in the definitions of some of the

control primitives. For example, to perform a CIA one must perform a CIA'. Hence, the essence of the control will not be explained. These issues will be discussed further in section 5.3.2, where we give a justification for the formal definition as a whole. Instead, the evaluators will synchronize themselves using the control primitives TSET and CLEAR. Although this will lead to direct circularities in the definitions of these operations, the circularities are not as suspect since the operations are themselves intuitively acceptable. They can be implemented in one machine instruction.

Synchronization is required by the evaluators:

- (1) to insure single access to the control interpreter path,
- (2) to insure that a path is evaluated by only one evaluator at a time,
- (3) to insure that the environment of a path is modified by only one evaluator at a time,
- (4) to detect and process external interrupts.

The activation record of each path contains a field MOD, of mode INT, which is used to provide synchronization in the first three cases above. Whenever a path is active, its MOD field has been TSET by the corresponding evaluator. Hence, if a path P is active, then TSET(P.MOD) returns FALSE. In particular, this is true of the CI path. Hence,

the evaluators may determine if the CI path is being evaluated.

Any control primitive that modifies the environment of a path other than the one in which it is called, TSETs the MOD field of the path in question. Hence, if TSET(P.MOD) returns FALSE then P is either active or being modified by a control primitive.

External interrupts are sent to an evaluator by an external processor or by another evaluator. To 'send' an interrupt, a structure associated with the evaluator is modified to indicate the type of interrupt and its priority level and then a flag is set to indicate that an interrupt is pending. The evaluator checks this flag at points during the evaluation at which it is convenient to allow an interrupt. If the flag is set, then it determines whether or not a response is to be initiated by examining the associated interrupt structure. Synchronization is required to insure that the evaluator and the interrupt generator do not modify the structure at the same time. A processor-interrupt-lock is associated with each evaluator. An evaluator TSETs this lock before accessing its interrupt structure. If the lock is set, then the evaluator goes into a busy wait upon the lock.

*

Justification for this model of interrupts will be given in section 5.3.2.

2. THE EL1 EVALUATOR

In this section we present the definition of the sub-evaluators for the language constructs of EL1. For each construct, examples of its use are given both in concrete and abstract syntax representation. A sub-evaluator is specified by a labelled sequence of statements. ^{*} Global constants are identifiers whose values are accessible to the NPROC evaluators but are not modified by them. Global variables are identifiers whose values are modified by the evaluators to effect inter-evaluator communication. All modes introduced are assumed to be global constants.

The EL1 evaluator presented here is similar to the evaluator in [Weg70], but has been updated to reflect changes in the language which have been included in a current implementation [Weg72]. The major difference between the two is in the method used to handle evaluator recursion, c.f. 4.1.2.

The first subsection describes evaluator initialization and the use of each local variable defined by an evaluator.

*

The complete definition of an evaluator is the concatenation of the sub-sections 4.2.i Evaluators (for i between 1 and 12) and 4.3.i Evaluators (for i between 1 and 15).

2.1 Declarations and Initialization

Global Constants

NPROC	;NT The number of processors;
PRO\PRO\FORM	;NT "PRO\PRO" response form;
TIMER\FORM	;NT "TIMER" response form;
NPALEV	;NT The number of processor interrupt levels;
NPROLEV	;NT The number of path interrupt levels;

Global Variables

PCIAR	;NT PTR to CI path;
PRO\PATH	;NT A ROW(NPROC,ARPTR) which specifies the assignment of processors to paths;
IDLE	;NT A ROW(NPROC,ARPTR) which specifies the idle path for each processor;
INIT\STATE	;NT A ROW(NPROC,SYMBOL) which is used to coordinate the initialization;

Modes

```

FORM <- PTR(INT, ATOM, DTPR, DDB, REF);
MODE <- PTR(DDE);
DDB <- STRUCT(CLASS:SYMBOL,
              D:PTR(DDE,
                  ROW(STRUCT(SYM:SYMBOL,TYPE:MODE)),
                  ROW(MODE),
                  STRUCT(TYPE:MODE,LENGTH:INT))),
              .
              .
              .
              );

```

NT See section 4.5 for a discussion
of the fields of a DDB;

NT See [Weg70] for a complete discussion
of modes in EL1;

DTPR <- STRUCT(CAR:FORM, CDR:FORM);

NAME\STACK <- ROW(STRUCT(NAME:SYMBOL, VALUE:REF));

VALUE\STACK <- STACK;

CONTROL\STACK <- STACK;

NSPTR <- PTR(NAME\STACK);

VSPTR <- PTR(VALUE\STACK);

CSPTR <- PTR(CONTROL\STACK);

ARPTR <- PTR(ACTRC);

ACTRC <-
STRUCT(NS:NSPTR,
VS:VSPTR,
CS:CSPTR,
NP:INT,
VP:INT,
CP:INT,
CIA\FN:REF,
CIA\ARG:REF,
CIA\RESULT:REF,
STKEFLG:BOOL,
ELGFIG:BOOL,
DORMANT:BOOL,
MOD:INT,
SPATH:BOOL,
IFLG:BOOL,
DS:ARPTR,
DSN:INT,
DSC:INT,
DSV:INT,
LBRO:ARPTR,
PLEV:ARPTR,
LASTSON:ARPTR,
LOWCP:INT,
NEXT:ARPTR,
TERMINATION\FORM:FORM,
TICKS\LEFT:INT,
USER\AR:ARPTR,
PRO:INT,
INPROI:INT,
INTINFO:ITE);

Evaluator

```

EVALUATOR <-
  EXPR(PROCNUM:INT, INITCI:BOOL, PROG:FORM; NONE)
  BEGIN

    NT      The following variables are used
            in intra-path evaluation;

    DECL F:FORM;
    DECL EVRES:REF;
    DECL NS:PTR(NAME\STACK);
    DECL VS:PTR(VALUE\STACK);
    DECL CS:PTR(CONTROL\STACK)
            BYVAL ALLOC(CONTROL\STACK SIZE 3);
    DECL RESULT\SLOT, AUX\RESULT\SLOT:PTR(VALUE\STACK)
            BYVAL ALLOC(VALUE\STACK SIZE 1);
    DECL NP, VP, CP, RSP, ARSP:INT;

    NT      The following variables are used in
            inter-path evaluation;

    DECL PATH:ARPTR;
    DECL SPATH,IFLG:BOOL;

    NT      The following variables are used as temps
            by the evaluator;

    DECL Q, P:ARPTR;
    DECL L, N, M, NAME\INDEX:INT;
    DECL S, TEMP:FORM;
    DECL B:BOOL;

    N <- 1;
    CALL EVGETPATH1; NT P points to a new path;
    IDLE[PROCNUM] <- P ;NT Indicate idle path created;
    INITCI -> GOTO INIT1;
    TSET(P.MOD);
    PRO\PATH[PROCNUM] <- P;
    INSTALL\STATE(P);
INIT2:  INIT\STATE[PROCNUM] # "CIREADY" -> GOTO INIT2
    CALL INIT\INTERRUPTS;
    INIT\STATE[PROCNUM] <- "PROREADY";

    NT IDLE!;

    DOIDLE: F <- QUOTE(WHILE TRUE DO NOTHING);
    CALL EVAL\FORM;

    INIT1:  PUSHC("DOIDLE",P);
            CALL EVGETPATH1 ;NT Create CI path;
            PCiar <- P;

```

```

TSET(PCIAR.MOD);
INSTALL\STATE(PCIAR);
INSTALL\GLOBAL\ENV()
      NT Install initial top level
        bindings for paths;
CALL INIT\INTERRUPTS;
FOR I <-1, ..., NPROC DO
  INIT\STATE[I] <- "CIREADY";
FOR I<-1, ..., NPROC DO
  BEGIN
    I = PROCNUM => NOTHING;
    L: INIT\STATE[I] # "PROREADY" -> GOTO L
  END;

NT Other processors are ready;

PUSHN("IDLE\PATHS",PUSHV(IDLE));
PUSHN("PROC",PUSHV(PROG));

F<-QUOTE(INIT\CI(IDLE\PATHS,PROG));
CALL EVAL\FORM; NT Initialize CI;

```

```

INIT\INTERRUPTS:
  F <- QUOTE(ENABLE\PRO("PRO\PRO", 1, PRO\PRO\FORM));
  CALL EVAL\FORM;
  F <- QUOTE(ENABLE\PRO("TIMER",2, TIMER\FORM));
  CALL EVAL\FORM;
  RETURN;

```

Discussion

The arguments to an evaluator specify its number ($1 \leq \text{PROCNUM} \leq \text{NPROC}$), a boolean which indicates whether or not the evaluator is to initialize the control interpreter path, and a form which is the program to be evaluated. Hence, assuming that PROGRAM is to be evaluated, the NPROC evaluators are initialized as follows:

```

EVALUATOR(1,TRUE,PROGRAM) ;
    EVALUATOR(2) ;
    .
    .
    EVALUATOR(NPROC-1) ;
    EVALUATOR(NPROC)

```

where ':' indicates that the evaluators execute the procedure calls simultaneously.

Each evaluator I (I//1) creates a path; enables itself for "PRO\PRO" and "TIMER" interrupts; and then idles. Evaluator 1 creates both its idle path and the control interpreter path; installs the 'top-level' environment (i.e. initializes top level bindings for all of the control subroutines, built-in functions, etc.); enables itself for interrupts; waits for the other processors to complete their initialization; binds the vector of idle paths and the program to be evaluated to variables in the CI's environment and then evaluates the procedure call INIT\CI(IDLE\PATHS,PROC) in the CI environment.

INIT\CI defines the variables to be used by the CI, creates a path in which the program is to be evaluated, and then calls C\I to commence path scheduling, c.f. Appendix 3.

*

Note that the 'top-level' environment of the evaluators is distinct from the 'top-level' environment seen by the paths.

The declared variables of an evaluator may be divided into three classes according to their use.

The first set of variables are used for intra-path evaluation. NS, VS, and CS point to the name, value, and control stacks of the path which is being evaluated. NP, VP, and CP index the top element of the three stacks. F specifies the current form which is being evaluated. EVRES is used to point to the value obtained by the evaluation of F. The value specified by EVRES may be in the heap, on the value stack of some path, or in the RESULT\SLOT.*

The second set of variables are used for inter-path evaluation. PATH specifies the path which is currently being evaluated. SPATH indicates whether or not PATH is a supporting path. If IFLG is TRUE, then PATH is currently evaluating a path or processor interrupt response.

The third set of variables is used locally in various parts of the evaluator.

*

The RESULT\SLOT (AUX\RESULT\SLOT) is used by the evaluator to hold the result produced by the evaluation of a language construct in certain cases. A value contained in the result\slot is called a pure-value.

2.2 Form

Modes

```
ATOM<-STRUCT(PRINT\NAME:PTR(String),TLB:REF);
SYMBOL<-PTR(ATOM);
STRING<-ROW(CHAR);
```

Evaluator

EVAL\FORM: ;NT F is the form to be evaluated;

```

F=NIL => RETURN\NOTHING;
M <- MVAL(F);
M = ATOM -> GOTO EVSYM;
M = DTPR -> GOTO EVDTPR;
BEGIN
    M = INT => EVRESULT(VAL(F),INT);
    M = REF => EVRESULT(VAL(F),MVAL(VAL(F)));
    M = DDB => EVRESULT(CONST(MODE LIKE F),MODE)
END;

RETURN:

```

```

EVSYM:  NAME\INDEX <- FIND\NAME(NS,NP,F);
        EVRES <- [ ) NAME\INDEX # 0 => NS[NAME\INDEX].VALUE;
                VAL(F).TLB ( );
        RETURN;

```

Auxiliary Functions

```

EVRESULT <-
  EXPR(PRES:ANY, RESMODE:MODE; REF)
  BEGIN
    DECL TEMP:REF;
    DECL TEMP1:INT;
    BEGIN
      MD(PRES) = REF AND
        INSTACK(PRES, 0, RSP, RESULT\SLOT) =>
          BEGIN
            TEMP <- RESULT\SLOT;
            RESULT\SLOT <- AUX\RESULT\SLOT;
            TEMP1 <- RSP;
            AUX\RESULT\SLOT <- TEMP;
            ARSP <- TEMP1
          
```

```

                END
            END;
            FLUSH(RESULT\SLOT, 0); RSP <- 0;
            EVRES <- PUSH(RPRES, RESMODE);
            EVRES
        END;

FIND\NAME <-
    EXPR(STACK:PTR(NAME\STACK),
        HIGH\INDEX:INT,
        NAME:SYMBOL; INT)
    BEGIN
        DECL H:INT BYREF HIGH\INDEX;
        DECL RESULT:INT;
        FOR I <- H,H-1 ... , 1 TILL RESULT # 0 DO
            BEGIN
                STACK[I].NAME = NAME ->
                RESULT <- I
            END;
        RESULT
    END;

```

Discussion

EVAL\FORM performs the evaluation of a form F based upon the mode M of the object referenced by F. If VAL(F) is atomic then EVRES is set to the value of the SYMBOL in the current environment or the top level binding of the SYMBOL. If it is a DTPR then control is transferred to the list structure evaluator. If it is an integer then EVRES is set to a copy of the integer which is pushed onto the RESULT\SLOT. Note that REFs are used to specify constants in the list structure representation of an MPEL1 program (e.g. TRUE, FALSE, 'C, etc.) If VAL(F) is a DDB (data-definition- block) then EVRES is set to a pointer to

*

The abstract syntax representation of identifiers with the same spelling are pointers to a unique ATOM.

the DDB (i.e. a mode.)

EVRESULT is used to push an object onto the RESULT\SLOT. The second argument to EVRESULT specifies the mode of the object. If the first argument is a REF, then the object to be copied is the VAL of the argument. Otherwise, the first argument specifies the object to be copied.

2.3 List Structure

Evaluator

```

EVDTPR:
  S <- F.CAR;
  MVAL(S) # ATOM -> GOTO APPLY;
  S = "EXPR!" -> GOTO EVEXPR;
  S = "BLOCK!" -> GOTO EVBLOCK;
  S = "IF!" -> GOTO EVIF;
  S = "CLAUSE!" -> GOTO EVCLAUSE;
  S = "FOR!" -> GOTO EVFOR;
  S = "SEL!" -> GOTO EVSEL;
  S = "SELQ!" -> GOTO EVSELQ;
  S = "DECL!" -> GOTO EVDECL;
  S = "LABST!" -> GOTO EVLABST;
  S = "<-" -> GOTO EVASSIGN;
  GOTO APPLY;

```

Discussion

If the CAR of the list is not a SYMBOL then the list structure is evaluated as a procedure application. If the CAR is a SYMBOL which indicates that the form is a language construct then control is transferred to the appropriate sub-evaluator. Otherwise, the list structure is evaluated as a procedure application.

2.4 Literal Procedure

Example

```

EXPR(X:INT; BOOL) [ ] X = 0 => TRUE ; FALSE (];
(EXPR! ((X INT BYREF)) BOOL
      (BLOCK! (CLAUSE! (= X 0) TRUE) FALSE))

```

Evaluator

```

EVEXPR: EVRESULT( CONST(FORM LIKE F), FORM);
RETURN;

```

Discussion

The value of a literal procedure is a pointer to the procedure.

2.5 Block

```

BEGIN B1 => 1; 2 END;
(BLOCK! (CLAUSE! B1 1) 2)
[ ] DECL X:INT ; FOO(X) (];
(BLOCK! (DECL! (X) INT) (FOO X))

```

Modes

```

BLOCK\BLOCK <-
  STRUCT(OLD\NP:INT,
         OLD\VP:INT,
         CUR\NP:INT,
         CUR\VP:INT,
         STATEMENT\LIST:FORM,
         RETURN:SYMBOL);

```

Evaluator

```

EVBLOCK: PUSHC(CONST(BLOCK\BLOCK OF
                    NP, VP, NP, VP, F.CDR, "RETBLOCK"));
EVRES <- NIL;
NT The value of a block is initially NOTHING;

```

```

EVBLK1:
  TOPC1.STATEMENT\LIST = NIL -> RETURN;
  CALL ALLOW\INTERRUPT;
  F <- TOPC1.STATEMENT\LIST;
  TOPC1.STATEMENT\LIST <- F.CDR;
  F <- F.CAR;
  CALL EVAL\FORM;
  GOTO EVBLK1;

```

```

RETBLOCK:
  MVAL(EVRES) = LABEL -> ERROR("illegal\result");
  NOT PURE\VALUE ->
    BEGIN
      INSTACK(EVRES, TOPC1.OLD\VP, VP, VS) ->
        EVRESULT(EVRES, MVAL(EVRES))
    END;
  NP <- TOPC1.OLD\NP;
  FLUSH(VS, TOPC1.OLD\VP);
  POPC1\RETURN;

```

Discussion

A block is evaluated by evaluating each statement on its statement list. Since the initial statements of a block may be declarations which add to the identifier environment it is necessary to record in the BLOCK\BLOCK the indices of

NP and VP which specify the environment in which the statements of the block are being evaluated. To allow for external interrupts, a call to ALLOW\INTERRUPT is made before the evaluation of each statement.

When the last statement of the block is evaluated, control is transferred to RETBLOCK. If the last value computed in the block is not a pure-value and it exists in the portion of the stack environment of the path which will be deleted upon block exit, then the value is copied into the result slot. The name and value stacks are flushed back to the level they were at before the block was entered, the BLOCK\BLOCK is removed from the control stack, and control returns to the caller of EVAL\FORM.

The value of a block is the value returned by the last statement executed.

2.6 Declaration

Examples

```
DECL X, Y:INT;
(DECL! (X Y) INT)
DECL Y:INT BYREF Z;
(DECL! (Y) INT BYREF Z)
```

```

DECL X:INT BYVAL 3;
(DECL! (X) INT BYVAL 3)
[] X; L: FOO ();
(BLOCK! (DECL! (L) LABEL
          (LABST! L FOO)) X (LABST! L FOO))

```

Modes

```

DECL\BLOCK <-
  STRUCT(ID\LIST:FORM,
         TYPE:FORM,
         INITD:FORM,
         EXP\MODE:MODE);

LABEL <- STRUCT(CPINDEX:INT, ST\LIST:FORM, PATH:ARPTR);

```

Evaluator

```

EVDECL:MVAL(TOPC2) # BLOCK\BLOCK ->
  ERROR("illegal\declaration");
PUSHC(CONST(DECL\BLOCK OF
            CADR(F), CADDR(F), F.CDR.CDR.CDR));

```

EVDECL1:

```

F <- TOPC1.TYPE;
CALL EVAL\FORM;
CHECKM(MODE);
TOPC1.EXP\MODE <- VAL(EVRES);
AND(TOPC1.INITD # NIL,
    TOPC1.INITD.CAR = "LABST!",
    VAL(EVRES) = LABEL) -> GOTO DECL\LABEL;
TOPC1.INITD = NIL -> GOTO DECL\NO\INIT;
F <- CADR(TOPC1.INITD);
CALL EVAL\FORM;
MVAL(EVRES) # TOPC1.EXP\MODE ->
  BEGIN
    COMPATIBLE(TOPC1.EXP\MODE, EVRES) ->
      EVRESULT(EVRES, TOPC1.EXP\MODE)
  END;

```

```

NOT PURE\VALUE AND TOPC1.INITD.CAR = "BYREF" ->
  GOTO DECL\BYREF;
PUSHN(TOPC1.ID\LIST.CAR, PUSHV(EVRES));
GOTO DECL\LOOP;

```

```

DECL\BYREF:
  PUSHN(TOPC1.ID\LIST.CAR, EVRES);

```

```

GOTO DECL\LOOP;

DECL\NO\INIT:
  TOPC1.EXP\MODE=LABEL OR
    TOPC1.EXP\MODE.CLASS="GENERIC"
    -> ERROR("illegal\declaration");
  PUSHN(TOPC1.ID\LIST.CAR, GENV(TOPC1.EXP\MODE));
  GOTO DECL\LOOP;

DECL\LABEL:
  PUSHN(TOPC1.ID\LIST.CAR,
    PUSHV(CONST(LABEL OF CP - 2,
      TOPC1.INITD, PATH)));

DECL\LOOP:
  TOPC3.CUR\NP <- NP;
  TOPC3.CUR\VP <- VP;
  TOPC1.ID\LIST <- TOPC1.ID\LIST.CDR;
  TOPC1.ID\LIST # NIL -> GOTO EVDECL1;
  POPC(2); NT Pop DECL\BLOCK and CALL from EVBLOCK;
  EVRES <- NS[NP].VALUE;
  GOTO EVBLK1;

```

Discussion

A declaration may only appear at the statement level of a block. The evaluation of a declaration results in the addition of one or more identifiers to the environment. The CUR\NP and CUR\VP fields of the BLOCK\BLOCK of the block are updated to reflect this addition.

For each identifier on the identifier list the following actions are performed. The type field is evaluated to produce a mode which is saved in the EXP\MODE component of the DECL\BLOCK. If the mode is LABEL then it is treated specially (see below). If the INITD field is NIL, then the identifier is bound to a default value on the value stack for the mode EXP\MODE. If it is not NIL, then

the initialization form is evaluated. If the initialization is to be BYREF and the result of evaluation is not a pure-value then the identifier is bound directly to the result. Otherwise, the identifier is bound to a copy of the result, which is pushed onto the value stack.

If the EXP\MODE is LABEL, then the identifier is bound to an entry on the value stack which specifies the BLOCK\BLOCK with which the label is associated, the sub-list of the statement list of the block starting with the labelled statement, and the current path.

The value of a declaration is the value associated with the last identifier bound on the name stack.

2.7 Conditional

Example

```
[ ] B -> FOO(A,B) ; A => B ; FALSE ( );
(BLOCK! (IF! B (FOO A B)) (CLAUSe! A B) FALSE)
```

Mode

```
COND\BLOCK <- STRUCT(LHSF:FORM, RHSF:FORM);
```

Evaluator

```

EVIF: MVAL(TOPC2) # BLOCK\BLOCK ->
      ERROR("illegal\conditional");
      PUSHC(CONST(COND\BLOCK OF CADR(F), CADDR(F)));
      F <- TOPC1.LHSF;
      CALL EVAL\FORM;
      CHECKM(BOOL) AND VAL(EVRES) = TRUE -> GOTO EVIF1;
      POPC(1);
      RETURN\NOTHING;

```

```

EVIF1:
      F <- TOPC1.RHSF;
      CALL EVAL\FORM;
      POPC1\RETURN; NT Return to EVBLOCK loop;

```

```

EVCLAUSE:
      MVAL(TOPC2) # BLOCK\BLOCK ->
      ERROR("illegal\conditional");
      PUSHC(CONST(COND\BLOCK OF CADR(F), CADDR(F)));
      F <- TOPC1.LHSF;
      CALL EVAL\FORM;
      CHECKM(BOOL) AND VAL(EVRES) = TRUE ->
      GOTO EVCLAUSE1;
      POPC(1);
      RETURN\NOTHING;

```

```

EVCLAUSE1:
      F <- TOPC1.RHSF;
      CALL EVAL\FORM;
      POPC(2); NT Flush COND\BLOCK
               and CALL from EVBLOCK;
      RETURN; NT Exit block;

```

Discussion

A conditional may only appear at the statement level of a block.

If the LHSF of an IF! form evaluates to TRUE, then the RHSF is evaluated and control returns to the EVBLOCK loop. If the LHSF of a CLAUSE! form evaluates to TRUE then the RHSF is evaluated and the result is taken as the value of

the block. In either case, if the LHSF evaluates to FALSE, then the value of the conditional is NOTHING.

2.8 Selection

Examples

```
X[3]
(SEL! X 3)
NS[NP].VALUE
(SELQ! (SEL! NS NP) VALUE)
```

Mode

```
SEL\BLOCK <-
  STRUCT(OBJF:FORM,
        SEL\FORM:FORM,
        OBJ:REF,
        INDEX:INT,
        SAVE\FLAG:BOOL);
```

Evaluator

```
EVSEL:PUSHC(CONST(SEL\BLOCK OF CADR(F), CADDR(F)));
  F <- TOPC1.OBJF;
  CALL EVAL\FORM;
  MVAL(EVRES).CLASS = "PTR" -> DEREf(EVRES);
  SAVE\VAL(); NT Save object on value-stack
    if pure-value;
  F <- TOPC1.SEL\FORM;
  CALL FVAL\FORM;
BEGIN
  MVAL(EVRES) = INT => TOPC1.INDEX <- VAL(EVRES);
  CHECKM(SYMBOL);
  TOPC1.INDEX <-
    SELECTOR\INDEX(MVAL(TOPC1.OBJ), VAL(EVRES));
END
```



```

EVSEL1:  TOPC1.INDEX LE 0 OR
          TOPC1.INDEX GT LENGTH(VAL(TOPC1.OBJ)) ->
          ERROR("selection\fault");
EVRES <-SELECT(TOPC1.OBJ, TOPC1.INDEX);
TOPC1.SAVE\FLAG -> GOTO UNSAVE\VAL;
POPC1\RETURN;

```

```

UNSAVE\VAL:
  EVRESULT(EVRES, MVAL(EVRES));
  POPV(1); NT Pop saved object off of value stack;
  POPC1\RETURN;

```

```

EVSELQ:
  PUSHC(CONST(SEL\BLOCK OF CADDR(F), CADDR(F)));
  F <- TOPC1.OBJF;
  CALL EVAL\FORM;
  MVAL(EVRES).CLASS = "PTR" -> DEREf(EVRES);
  SAVE\VAL();
  TOPC1.INDEX <-
    SELECTOR\INDEX(MVAL(TOPC1.OBJ),
                   TOPC1.SEL\FORM);
  GOTO EVSEL1;

```

Auxiliary Functions

```

SAVE\VAL <-
  EXPR(; NONE)
  BEGIN
    PURE\VALUE =>
      BEGIN
        TOPC1.OBJ <- PUSHV(EVRES);
        TOPC1.SAVE\FLAG <- TRUE
      END;
    TOPC1.OBJ <- EVRES
  END;

```

```

SELECTOR\INDEX <-
  EXPR(M:MODE, S:SYMBOL; INT)
  BEGIN
    DECL L:INT;
    M.CLASS # "STRUCT" =>
      ERROR("selection\fault");
    FOR I <- 1, ..., LENGTH(VAL(M.D)) TILL L GT 0 DO
      [ ] M.D[I].SYM = S => L <- I [ ];
    L
  END;

```

Discussion

Two types of selection forms are defined for compound objects, namely selection (SEL!) and selection-quoted (SELQ!). In either case, the OBJF is evaluated. If the result is a pointer, it is dereferenced to produce a non-pointer value. If the result is a pure value then it is saved on the value stack.

SEL! and SELQ! differ in the method used to obtain the index of the component to be selected.

SELQ! calls SELECTOR\INDEX to obtain, from the mode of the object, the index associated with the symbolic field name specified by SEL\FORM. SEL! evaluates the SEL\FORM. If the result is an integer then it uses it as the index. If the result is a SYMBOL, then it calls SELECTOR\INDEX to obtain the index. The primitive procedure SELECT is called to select the appropriate component of the object. SELECT returns a pointer to the selected component.

The result of a selection form is the component of the object.

*

VAL is applied repeatedly until a non-pointer object is obtained. Selection is the only language construct in which pointer coercion is automatic.

2.9 Assignment

Examples

```

X ← 1
(← X 1 )
S ← Y.CAR
(← S (SELQ! Y CAR))

```

Modes

```

ASSIGN\BLOCK ←
  STRUCT(LHSF:FORM,
         RHSF:FORM,
         OBJ:REF,
         SAVE\FLAG:BOOL);

```

Evaluator

```

EVASSIGN:
  PUSHC(CONST(ASSIGN\BLOCK OF CADR(F), CADDR(F)));
  F ← TOPC1.LHSF;
  CALL EVAL\FORM;
  SAVE\VAL();
  F ← TOPC1.RHSF;
  CALL EVAL\FORM;
  NOT COMPATIBLE(MVAL(TOPC1.OBJ), EVRES) ->
    ERROR("assign\error");
  ASSIGN(TOPC1.OBJ, EVRES);
  TOPC1.SAVE\FLAG -> GOTO UNSAVE\VAL;
  EVRES ← TOPC1.OBJ;
  POPC1\RETURN;

```

Discussion

The LHSF is evaluated first. If the result is a pure-value then it is saved on the value stack. The RHSF is then evaluated. If the 2 objects obtained are compatible

then the primitive function ASSIGN is used to perform the mode-dependent assignment. The value of an assignment form is the object specified by OBJF after the assignment has been completed, unless the LHSF was a pure value in which case the value of the assignment is a copy of the modified LHSF.

2.10 Iteration

Examples

```
FOR I <- 1, ..., N DO SUM <- SUM + 1;
(FOR I 1 NIL N NIL (<- SUM (+ SUM 1 )))
FOR I <- 1,3, ..., K TILL P(I) DO
    [ ] B => Q(X) ; T(X) [ ];
(FOR I 1 3 K (TILL . (P I))
    (BLOCK! (CLAUSE! B (Q X)) (T X)))
FOR I<- 1,3, ..., N WHILE B DO (FOO(X));
(FOR I 1 3 N (WHILE . B) (FOO X))
```

Modes

```
FOR\BLOCK <-
  STRUCT(OLD\NP:INT,
    OLD\VP:INT,
    NAME:FORM,
    INITF:FORM,
    STEPF:FORM,
    LIMITF:FORM,
    TESTF:FORM,
    COND\FLAG:BOOL,
    BODY:FORM,
    STEP:INT,
```

```

LIMIT:INT,
RETURN:SYMBOL);

```

Evaluator

```

EVFOR:
  PUSHC(CONST(FOR\BLOCK OF
    NP,
    VP,
    {F<-F.CDR}.CAR,
    {F<-F.CDR}.CAR,
    {F<-F.CDR}.CAR,
    {F<-F.CDR}.CAR,
    {F<-F.CDR}.CAR,
    BEGIN
      F.CAR = NIL => FALSE;
      F.CAR.CAR = "TILL" => TRUE;
      FALSE
    END,
    {F<-F.CDR}.CAR,
    1,
    0,
    "RETFOR"));
  PUSHN(TOPC1.NAME, GENV(INT));
  F <- TOPC1.INITF;
  CALL EVAL\FORM;
  CHECKM(INT);
  NSV1 <- VAL(EVRES);
  F <- TOPC1.STEPF;
  F = NIL -> GOTO EVFOR5;
  CALL EVAL\FORM;
  CHECKM(INT);
  TOPC1.STEP <- VAL(EVRES) - NSV1;
EVFOR5:
  F <- TOPC1.LIMITF;
  F = NIL -> GOTO EVFOR1;
  CALL EVAL\FORM;
  CHECKM(INT);
  TOPC1.LIMIT <- VAL(EVRES);
EVFOR1:
  TOPC1.TESTF # NIL -> GOTO EVFOR3;
EVFOR2:
  CALL ALLOW\INTERRUPT;
  SIGN(TOPC1.STEP) * (NSV1 - TOPC1.LIMIT) GT
    0 -> GOTO ENDFOR;
  F <- TOPC1.BODY;
  CALL EVAL\FORM;
  NSV1 <- NSV1 + TOPC1.STEP;
  GOTO EVFOR2;

```

```

EVFOR3:
  CALL ALLOW\INTERRUPT;
  SIGN(TOPC1.STEP) * (NSV1 - TOPC1.LIMIT) GT
    0 -> GOTO ENDFOR;
  F <- TOPC1.TESTF.CDR;
  CALL EVAL\FORM;
  CHECKM(BOOL);
  TOPC1.COND\FLAG = VAL(EVRES) -> GOTO ENDFOR;
  F <- TOPC1.BODY;
  CALL EVAL\FORM;
  NSV1 <- NSV1 + TOPC1.STEP;
  GOTO EVFOR3;
ENDFOR:
  RETURN\NOTHING;

RETFOR:
  NP <- TOPC1.OLD\NP;
  FLUSH(VS, TOPC1.OLD\VP);
  POPC1\RETURN;

```

Auxiliary Function

```

SIGN <-
  EXPR(N:INT; INT)
  BEGIN
    N = 0 => 0;
    N GT 0 => 1;
    -1
  END;

```

Discussion

There are two types of iteration, namely, iteration without-test and iteration with-test. In either case, a name-stack entry is made for the iteration variable. The initial value for the iteration variable is obtained by evaluating INITF. If STEPF is non-null then it is evaluated and the iteration STEP is taken to be the difference between it and the result of evaluating INITF, otherwise the STEP is defaulted to 1. If the LIMITF is non-null, it is evaluated

to yield the iteration limit, otherwise the LIMIT is defaulted to 0. The values for STEP and LIMIT are saved in the FOR\BLOCK.

For an iteration without test (TESTF=NIL), the iteration body is evaluated 0 or more times until the iteration variable exceeds the LIMIT.

For an iteration with test (TESTF#NIL), the iteration body is evaluated 0 or more times until either the iteration variable exceeds the limit or the result of evaluating TESTF.CDR is equal to the COND\FLAG.*

Since the evaluation of an iteration form adds an identifier to the environment, return from the evaluation must be via the return component in the FOR\BLOCK.

To allow for external interrupts, a call to ALLOW\INTERRUPT is made before each evaluation of the body.

The result of an iteration form is NOTHING.

*
COND\FLAG is set to TRUE or FALSE as TESTF.CAR equals "TILL" or "WHILE", respectively.

2.11 Procedure Application

Examples

```

FOO(A, B, C + D)
(FOO A B (+ C D))
(EXPR(X:INT,Y:INT BYVAL;INT)(X<-Y))(A,E)
((EXPR! ((X INT BYREF)(Y INT BYVAL))
  INT (<- X Y)) A B)

```

Modes

```

FN\BLOCK <-
  STRUCT(OLD\NP:INT,
    OLD\VP:INT,
    ARG\LIST:FORM,
    RESULT\TYPE:MODE,
    PROC:REF,
    TYPE:SYMBOL,
    NAME:SYMBOL,
    RETURN:SYMBOL,
    ENTERED:BOOL);

BINDF\BLOCK <-
  STRUCT(ACTUAL\LIST:FORM,
    FORMAL\LIST:FORM,
    EXP\MODE:MODE,
    BCLASS:SYMBOL);

CEXP <-
  STRUCT(FORMAL\LIST:FORM,
    BODY:ROW(INT),
    RESULT\TYPE:FORM);

CSUBR <-
  STRUCT(FORMAL\LIST:FORM,
    BODY:SYMBOL,
    RESULT\TYPE:MODE);

```

Evaluator

```
APPLY:PUSHC(CONST(FN\BLOCK OF
```

```
NP,  
VP,  
F.CDR,  
NIL,  
NIL,  
NIL,  
NIL,  
"RETFN",  
FALSE));
```

```
F <- F.CAR;  
MVAL(F) = ATOM -> TOPC1.NAME <- F;  
NT Save name of procedure;  
CALL APPLY1;  
MVAL(TOPC2) = PAP\BLOCK AND TOPC2.PATH # PATH ->  
GOTO DOPAP;
```

```
APPLY2:
```

```
TOPC1.ENTERED <- TRUE;  
TOPC1.TYPE = CEXPR -> GOTO APCEXPR;  
TOPC1.TYPE = CSUBR -> GOTO APCSUBR;  
F <- CADDR(TOPC1.BODY);  
CALL EVAL\FORM;
```

```
PROCRET:
```

```
PROC\EXIT(TOPC1.OLD\VP, TOPC1.RESULT\TYPE);  
RETURN;
```

```
APPLY1:
```

```
CALL EVAL\FORM;  
DEREF(EVRES);  
M <- MVAL(EVRES);  
NOT OR(M = CEXPR, M = CSUBR,  
M = DTPR AND EVRES.CAR = "EXPR!") ->  
ERROR("unbound\proc");  
TOPC2.TYPE <- M;  
TOPC2.PROC <- EVRES;  
CALL BINDF;  
TOPC2.TYPE = DTPR -> PUTNAMES(TOPC2.PROC.CDR.CAR);  
TOPC2.TYPE = CEXPR -> PUTNAMES(TOPC2.PROC.FORMALS);  
F <- [ ] TOPC2.TYPE = DTPR => CADDR(TOPC2.PROC);  
TOPC2.PROC.RESULT\TYPE [ ]  
CALL EVAL\FORM;  
CHECKM(MODE);  
TOPC2.RESULT\TYPE <- VAL(EVRES);  
CALL ALLOW\INTERRUPT;  
RETURN;
```

```
APCEXPR: XCT(TOPC1.PROC.BODY); NT execute code procedure;  
GOTO PROCRET;  
NT Result is set by code procedure
```


and is pointed to by EVRES;

APCSUBR: GOTO EVAL(TOPC1.PROC.BODY)

NT Transfer control to appropriate
point in evaluator;

RETFN: NP <- TOPC1.OLD\NP;
VP <- TOPC1.OLD\VP;
POPC1\RETURN;

BINDF:
PUSHC(CONST(BINDF\BLOCK OF
TOPC3.ARG\LIST,
BEGIN
TOPC3.TYPE = DTPR =>
CADR(TOPC3.PROC);
TOPC3.PROC.FORMALS
END));

BINDF3:
TOPC1.FORMAL\LIST = NIL -> GOTO ENDBINDF;
TOPC4.TYPE = CSUBR -> GOTO BINDF1;
F <- CADR(TOPC1.FORMAL\LIST.CAR);
CALL EVAL\FORM;
CHECKM(MODE);
TOPC1.EXP\MODE <- VAL(EVRES);

BINDF4:
TOPC1.BCLASS <- CADDR(TOPC1.FORMAL\LIST.CAR);
TOPC1.BCLASS = "UNEVAL" -> GOTO BINDUNEVALED;
TOPC1.BCLASS = "LISTED" -> GOTO BINDLISTED;
TOPC1.ACTUAL\LIST = NIL -> GOTO GENDEF;
F <- TOPC1.ACTUAL\LIST.CAR;
CALL EVAL\FORM;
BEGIN
TOPC1.EXP\MODE.CLASS = "GENERIC" =>
TOPC1.EXP\MODE <-
RESOLVE(TOPC1.EXP\MODE, MVAL(EVRES));
NOT COMPATIBLE(TOPC1.EXP\MODE, EVRES) ->
ERROR("type\fault")
END;
TOPC1.EXP\MODE # MVAL(EVRES) ->
EVRESULT(EVRES, TOPC1.EXP\MODE);
TOPC1.BCLASS = "BYREF" AND NOT PURE\VALUE ->
GOTO BINDBYREF;
PUSHN(NIL, PUSHV(EVRES)); NT Bind BYVAL;
GOTO BINDFLOOP;

BINDBYREF:
PUSHN(NIL, EVRES);

```

        GOTO BINDFLOOP;

BINDUNEVALED:
    TOPC1.EXP\MODE # FORM ->
        ERROR("mode\bind\class\mismatch");
    PUSHN(NIL, PUSHV(TOPC1.ACTUAL\LIST.CAR));
    GOTO BINDFLOOP;

BINDLSTD:
    CHECKM(FORM);
    PUSHN(NIL, TOPC1.ACTUAL\LIST);
    TOPC1.ACTUAL\LIST <- NIL;

BINDFLOOP:
    TOPC1.ACTUAL\LIST # NIL ->
        TOPC1.ACTUAL\LIST <- TOPC1.ACTUAL\LIST.CDR;
    TOPC1.FORMAL\LIST <- TOPC1.FORMAL\LIST.CDR;
    GOTO BINDF3;

ENDBINDF:
    POPC1\RETURN;

BINDF1:
    TOPC1.EXP\MODE <- CADR(TOPC1.FORMAL\LIST.CAR);
    GOTO BINDF4;

GENDEF:
    TOPC1.EXP\MODE = LABEL OR
    TOPC1.EXP\MODE.CLASS = "ONEOF" ->
        ERROR("illegal\binding");
    PUSHN(NIL, GENV(TOPC1.EXP\MODE));
    GOTO BINDFLOOP;

```

Auxiliary Functions

```

PUTNAMES <-
    EXPR(L:FORM BYVAL; NONE)
    BEGIN
        DECL N:INT BYVAL TOPC2.OLD\NP;
        FOR I <- N + 1, ... , NP DO
            [] NS[I].NAME <- L.CAR.CAR; L <- L.CDR (]
        END;

RESOLVE <-
    EXPR(U:MODE, R:MODE; MODE)
    BEGIN
        DECL FF:BOOL;
        U = ANY => R;
        U.CLASS # "GENERIC" => ERROR("resolve\error");
        FOR I <- 1, ... , LENGTH(VAL(U.D)) TILL FF DO

```

```

      [ ] U.D[I] = R => FF <- TRUE [ ];
      FF = TRUE => R;
      ERROR("resolve\error")
END;

```

```

PROC\EXIT <-
  EXPR(OLDVP:INT, EXPMODE:MODE; REF)
  BEGIN
    MVAL(EVRES) = LABEL => ERROR("illegal\result");
    BEGIN
      EXPMODE = NONE => EVRESULT(ALLOC(NONE), NONE);
      EXPMODE.CLASS = "GENERIC" =>
        EXPMODE <- RESOLVE(EXPMODE, MVAL(EVRES));
      NOT COMPATIBLE(EXPMODE, EVRES) =>
        ERROR("type\fault")
    END;
    OLDVP = VP => EVRES;
    INSTACK(EVRES, OLDVP, VP, VS) =>
      EVRESULT(EVRES, EXPMODE);
    MVAL(EVRES) = EXPMODE => EVRES;
    EVRESULT(EVRES, EXPMODE)
  END;

```

Discussion

A form (f a1 a2 ...an), where f does not specify that the form is a language construct is treated as a call on the procedure f with actual parameters a1 ,..., an.

Procedure application is carried out in five steps:

- (1) f is evaluated to obtain a procedure.
- (2) The formal parameters of the procedure are bound to the actuals a1 ,..., an.
- (3) The result type of the procedure is evaluated to obtain the EXPECTED\MODE of the call (i.e. the mode of the object which will be returned by the procedure.)
- (4) The procedure body is evaluated.

- (5) The procedure PROC\EXIT is called to check the mode of the result against the EXPECTED\MODE and to check whether the result exists in the portion of the stack environment which will be deleted upon procedure exit.

A procedure is either an explicit procedure, a code procedure, or a control subroutine.

An explicit procedure is one which is defined in EL1 and whose external representation is of syntactic type <exprnt>, c.f. Appendix 2.

A code procedure (CEXP) is one written in a language other than EL1. All (non-control) primitives (such as +, -, VAL, MD, CONST, ALLOC) are assumed to be defined as code procedures. The BODY component of a code procedure is a ROW(INT) which specifies the machine code to be executed.

A control subroutine (CSUBR) is one of the control primitives described in chapter 2. The body of a control subroutine specifies the point in the evaluator to which control is to be passed to perform the desired control action.

The formal parameters to a procedure are represented as a list of the form

((P1 MFORM1 BCLASS1) ... (Pn MFORMn BCLASSn))

where P_i is the name of the i th formal, $MFORM_i$ is a form

which is to be evaluated to obtain the mode of the i th formal^{*}, and BCLASS i must be one of the following SYMBOLs: BYVAL, BYREF, UNEVAL, or LISTED.

If an argument is passed BYVAL, then the formal is bound to a copy of the value of the corresponding actual. If an argument is passed BYREF, then the formal is bound to the result of evaluating the argument itself (unless the result is a pure-value.) An UNEVALed argument is bound to a pointer to the list structure for its corresponding actual (which is not evaluated). A LISTED argument is bound to a pointer to the remaining argument list. If the list of actuals is exhausted before all formals have been bound, then the remaining formals are bound to objects which are the default values for the corresponding modes.

All arguments are evaluated in the identifier environment which exists at the point at which the procedure call is made, hence the names of the formals are not put onto the name stack until all arguments are evaluated. The names of the formals of CSUBRs are never put on the stack.

The result-type is evaluated in an environment which includes the bindings of the formals.

*

The modes of the formals for CSUBRs are assumed to be implicit in the list structure. Hence, no evaluation is necessary.

At this point, a call to ALLOW\INTERRUPT is made to allow for external interrupts. In addition, a check is made to see if the body of the procedure is to be applied in the environment of another path (due to a call on PAP.) If so, control is transferred to DOPAP. Otherwise, the body of the procedure is evaluated.

Since a procedure application may add identifiers to the environment, return from the procedure application must be via the RETURN component of the FN\ELOCK.

2.12 Labelled Statement

Examples

```
L: X ← 1;
  (LABST! L ( ← X 1))
L1: L2: FOO(A,B);
  (LABST! L1 (LABST! L2 (FOO A B )))
```

Evaluator

```
EVLABST:
  F ← F.CDR.CDR.CAR;
  GOTO EVAL\FORM;
```

Discussion

The value of a labelled statement is the value obtained by evaluating the statement itself.

3. THE CONTROL PRIMITIVES

In this section we present the definitions of the bodies of the control subroutines. The definitions of the control primitives are installed as 'top-level' bindings by ^{*}INSTALL\GLOBAL\ENV as objects of mode CSUBR. The BODY component of a CSUBR specifies the label in the evaluator at which the body of the CSUBR is defined. Recall that calls upon the control primitives appear syntactically in EL1 programs as procedure calls (c.f. 4.2.11). Hence, when control is transferred to the body of the control subroutine, the arguments ^{**}for the call have been bound on the name stack of the path.

For each control primitive we present its definition as a CSUBR in the format of a procedure heading which specifies the modes and bind classes of its arguments, the mode of its result and the evaluator label at which its body is located.

*

Evaluator initialization, including the initialization of top-level bindings, is discussed in section 4.2.1

**

The body of a control subroutine references its arguments via the NOFIX operators NSV1, NSV2, etc., e.g. NSV1 is equivalent to VAL(NS[NP].VALUE) - the last argument passed to the control subroutine.

3.1 GET\PATH

Definition

```
GET\PATH ← CSUBR(N:INT;ARPTR) EVGETPATH;
```

Example

```
Q ← GET\PATH(3);
```

Mode

```
ENV\BLOCK ← STRUCT(OLD\NP:INT,  
                   OLD\VP:INT,  
                   RETURN:SYMBOL);
```

Global Constants

```
CI\PATH\FORM      ; NT "CI\PATH" response form;  
TIME\OUT\FORM     ; NT "TIME\OUT" response form;  
NSQUANT           ; NT Minimum size for NAME\STACK;  
VSQUANT           ; NT Minimum size for VALUE\STACK;  
CSQUANT           ; NT Minimum size for CONTROL\STACK;
```

Evaluator

```
EVGETPATH:
```

```
  N ← 1;  
  CALL EVGETPATH1;  
  RETURN\RESULT(P);
```

```
EVGETPATH1:
```

```
  P ← ALLOC(ACTRC);
```

```
  NT Initialize path level interrupt structure;
```

```
  P.INTINFO.CURLEV ← NPALEV + 1;  
  P.INTINFO.WAITLEV ← NAPLEV + 1;  
  P.INTINFO.RESP[1] ← CI\PATH\FORM;
```

```

P.INTINFO.TYPE[1] <- "CI\PATH" ;
P.INTINFO.RESP[2] <- TIME\OUT\FORM ;
P.INTINFO.TYPE[2] <- "TIME\OUT" ;

```

NT Initialize stacks ;

```

P.NS <- ALLOC(NAME\STACK SIZE N*NSQUANT);
P.VS <- ALLOC(VALUE\STACK SIZE N*VSQUANT);
P.CS <- ALLOC(CONTROL\STACK SIZE N*CSQUANT);

PUSHC(CONST(ENV\BLOCK OF 0, 0, "DELPTH"), P);
P.STKEFLG <- TRUE;
P.ELGFLG <- TRUE;

```

NT Initialize termination form;

```

P.TERMINATION\FORM <-
    QUOTE(CIA("DELETE\PATH", MYPATH));

```

RETURN;

NT Control underflow handler;

```

DELPTH: PUSHN("LAST\VALUE", PUSHV(EVRES));
F <- PATH.TERMINATION\FORM;
CALL EVAL\FORM;
ERROR("termination\error");

```

Discussion

GET\PATH creates a new path. The path's ACTRC is allocated in the heap. It is enabled for the path level interrupts "CI\PATH" and "TIME\OUT". The label (DELPTH) of a statement in the evaluator to which control is to be transferred upon control underflow is pushed onto the control\stack. The initial termination form for the path is set to be one which will call DELETE\PATH.

Control is transferred to DELPTH upon exit from the outermost procedure call which has been PAPed into the path.

The last value computed is bound to the name LAST\VALUE and the path's TERMINATION\FORM is evaluated. If the TERMINATION\FORM does not terminate the path, then an error occurs.

3.2 PAP, PAPQ, DPAP, DPAPQ

Definitions

```
PAP <- CSUBR(F:FORM, P:ARPTR; ARPTR) EVPAP;
PAPQ <- CSUBR(F:FORM UNEVAL, P:ARPTR; ARPTR) EVPAP;
DPAP <- CSUBR(F:FORM, P:ARPTR; ARPTR) EVDPA;
DPAPQ <- CSUBR(F:FORM UNEVAL, P:ARPTR; ARPTR) EVDPA;
```

Examples

```
PAPQ(FOO(A, B), P1)
DPAP(X, MDEP(GET\PATH(1)))
```

Mode

```
PAP\BLOCK <- STRUCT(PATH:ARPTR, DEPFLG:BOOL);
```

Evaluator

```
EVDPA: B <- TRUE; GOTO EVPAP1;
EVPAP: B <- FALSE;
EVPAP1: NSV1 = NIL OR NSV1 = PATH -> GOTO EVPAP3;
        EXISTS(NSV2,
                SEL!,
```

```

        SELQ!,
        FOR,
        DECL!,
        BLOCK!,
        <-,
        EXPR,
        IF!,
        CLAUSE!,
        LABST!) -> GOTO DOPAP1;
PUSHC(CONST( PAP\BLOCK OF NSV1, B));
F <- NSV2;
GOTO APPLY;

```

NT Begin the procedure application;

NT APPLY passes control to DOPAP just before
the procedure is applied;

```

DOPAP: CHECK\PATH(TOPC2.PATH);
      MOVE\ARGS(TOPC2.DEFPG);
      PUSHC("APPLY2", TOPC2.PATH);
      CLEAR(TOPC2.PATH.MOD);
      POPC1 ; NT Pop unused FN\BLOCK;
      P <- TOPC1.PATH;
      POPC1; NT Pop PAP\BLOCK;
      RETURN\RESULT(P);

```

NT Evaluate the form in the current
path's environment;

```

EVPAP3: F <- NSV2;
      CALL EVAL\FORM;
      RETURN\RESULT(NSV1);

```

NT Modify environment of target path so that
the form will be evaluated;

```

DOPAP1: CHECK\PATH(NSV1);
      PUSHC(NSV2, NSV1); NT Save form on CONTROL\STACK ;
      PUSHC("PAPF", NSV1);
      CLEAR(TOPC2.PATH.MOD);
      RETURN\RESULT(NSV1);

```

NT PAPF evaluates the form which is the top
element of the control stack;

```

PAPF:  F <- VAL(TOPC1);
      POPC1;
      GOTO EVAL\FORM;

```

Auxiliary Functions

```
CHECK\PATH <-
  EXPR(P:ARPTR; NONE)
  BEGIN
    NOT TSET(P.MOD) -> ERROR("path\mod");
    NOT P.STKEFLG -> ERROR1("no\stacks",P)
  END;
```

```
EXISTS <-
  EXPR(F:FORM, L:FORM LISTED; BOOL)
  BEGIN
    MVAL(F) # DTPR -> FALSE;
    INSET(F.CAR, L)
  END;
```

```
INSET <-
  EXPR(X:FORM, L:FORM; BOOL);
  BEGIN
    L = NIL => FALSE;
    X = L.CAR => TRUE;
    INSET(X, L.CDR)
  END;
```

```
MOVE\ARCS <-
  EXPR(DFLG:BOOL; NONE)
  BEGIN
    DECL P:ARPTR BYVAL TOPC2.PATH;
    DECL DD:ARPTR;
    DECL N:INT BYVAL TOPC1.OLD\NP;
    TOPC1.OLD\NP <- P.NP;
    TOPC1.OLD\VP <- P.VP;
    PUSHC(VAL(TOPC1), P);
    FOR I <- N + 1, ... , NP DO
      BEGIN
        DECL B:BOOL;
        MVAL(NS[I].VALUE) = LABEL =>
          PUSHN(NS[I].NAME, GENV(LABEL, P), P);
        B <-
          BEGIN
            DFLG AND (DD <- DDEP(PATH, P)) # NIL =>
              INSTACK(NS[I].VALUE, DD.DSV, VP, VS);

            NOT HEAP(NS[I].VALUE)
          END;
        PUSHN(NS[I].NAME,
          BEGIN
            B => PUSHV(NS[I].VALUE, P);
            NS[I].VALUE
```



```

                                END,
                                P)
      END
END;

```

Discussion

(D)PAP(Q) arranges for the evaluation of a form or the application of a procedure in the environment of another path. If NSV1 is null, or is equal to the current path, then the form NSV2 is simply evaluated in the current path's environment.

If NSV2 is not a procedure application, then the environment of NSV1 is modified, so that if control is passed to it, the form will be evaluated. This is accomplished by pushing the form and the evaluator label PAPF onto the control stack of NSV1.

If NSV2 is a procedure application, then a PAP\BLOCK is pushed onto the control stack of the current path and the procedure application is "evaluated". APPLY checks to see if the procedure application is to be applied in the current environment or not just before it evaluates the body of the procedure (i.e. after the actuals and result-type have been evaluated.) If APPLY detects a PAP\BLOCK immediately preceding the FN\BLOCK it has placed on the control stack, then it passes control to DOPAP in lieu of evaluating the procedure body.

MOVE\ARGS copies the arguments and FN\BLOCK into the environment of NSV1. The boolean argument specifies whether or not the PAP was a dependent one, i.e. whether arguments which exist in the accessible environment of NSV1 are to be passed directly or are to be copied.

DOPAP pushes the interpreter label APPLY2 onto the control stack of NSV1 so that if control is passed to NSV1, the body of the procedure will be evaluated.

The modification word in NSV1's ACTRC is TSET to insure that two evaluators do not simultaneously modify NSV1's environment.

The result of (D)FAP(Q) is NSV1.

3.3 PFETCH, PSTORE

Definitions

```
PFETCH <- CSUBR(NAME:SYMBOL, P:ARPTR BYVAL; ANY) EVPFETCH;
```

```
PSTORE <- CSUBR(VALUE:ANY,
                 NAME:SYMBOL,
                 P:ARPTR BYVAL) EVPSTORE;
```

Examples

```

PFETCH("X", P)

PSTORE(A+B, "Y", Q)

```

Evaluator

```

EVPFETCH: SETUP();
          NOT HEAP(EVRES) -> EVRESULT(EVRES, MVAL(EVRES));
          NSV1 # PATH -> CLEAR(NSV1.MOD);
          RETURN;

EVPSTORE: SETUP();
          NOT COMPATIBLE(MVAL(EVRES), NS3) ->
            ERROR1("assign\error", NS3);
          ASSIGN(EVRES, NS3);
          NSV1 # PATH -> CLEAR(NSV1.MOD);
          RETURN\NOTHING;

```

Auxiliary Function

```

SETUP <-
  EXPR(; NONE)
  BEGIN
    NSV1 = NIL -> NSV1 <- PATH;
    NSV1 # PATH AND NOT TSET(NSV1.MOD) ->
      ERROR("path\mod");
    NOT NSV1.STKEFLG -> ERROR("no\stacks", NS1);
    NAME\INDEX <-
      FIND\NAME(NSV1.NS,
        N <- [ ] NSV1 = PATH => NP; NSV1.NP [ ],
        NSV2);
    NAME\INDEX = 0 -> ERROR("no\binding");
    EVRES <- NSV1.NS[NAME\INDEX].VALUE
  END;

```

Discussion

PFETCH obtains the most recent binding of the identifier specified by NSV2 in the path specified by NSV1. If the identifier is not bound directly to an object in the heap, then the value of the identifier is copied into the RESULT\ SLOT.

PSTORE assigns NSV2 to the most recent binding of the identifier NSV2 in the path NSV1. The modes of the objects must be compatible for assignment.

In either case, an error occurs if there is no binding for NSV2 in path NSV1.

3.4 TSET, CLEAR

Definitions

```
TSET ← CSUBR(X:INT; BOOL) EVTSET;
CLEAR ← CSUBR(X:INT; NONE) EVCLEAR;
```

Evaluators

```
EVTSET: RETURN\RESULT(TSET1(NSV1), BOOL);
EVCLEAR: CLEAR1(NSV1);
          RETURN\NOTHING;
```


Discussion

TSET 'sets' the integer NSV1 and returns TRUE or FALSE as NSV1 was 'unset' or 'set' previously. The test-and-set is an indivisible operation. CLEAR 'unsets' the integer in a single indivisible operation.

3.5 MDEP, DEPENVDefinitions

```
MDEP <- CSUBR(P:ARPTR; ARPTR) EVMDEP;
DEPENV <- CSUBR(X:SYMBOL; ANY) EVDEPENV;
```

Examples

```
P <- MDEP(GET\PATH(1));
DECL X:INT BYREF DEPENV("WALDO");
```

Evaluators

```
EVMDEP: NSV1 = PATH OR NSV1 = NIL ->
        ERROR("dependency");
NOT TSET(NSV1.MOD) ->
        ERROR("path\mod");
NSV1.DS # NIL AND NSV1.DS # PATH ->
        ERROR1("dependency",NSV1);
NSV1.DS = PATH -> REM\DEPLIST(NSV1, PATH);
        NT Remove NSV1 from the
        list of dependents;
DDEP(NSV1, PATH) # NIL ->
        ERROR1("dependency",NSV1);
        NT If PATH is a dependent of NSV1 then
        a circular dependency will be created;
BEGIN
        (NSV1.DSN <- FIND\NENTRY()) = 0 =>
```

```

      BEGIN
        NSV1.DSN <- 0;
        NSV1.DSC <- 1;
        NSV1.DSV <- 0;
      END;
    FIND\CENTRY\VENTRY(NSV1.DSN, NSV1)
  END;
  ADD\DEPLIST(NSV1, PATH);
  CS[PATH.LOWCP]["RETURN"] <- "CHECK\SUPPORT";
    NT Smash RETURN component ;
  CLEAR(NSV1.MOD);
  RETURN\RESULT(NSV1);

```

```

CHECK\SUPPORT:
  NOT CHECK\LEV(CP - 1) -> ERROR("non\support");
  MVAL(TOPC1) = FN\BLOCK -> GOTO RETFN;
  MVAL(TOPC1) = BLOCK\BLOCK -> GOTO RETELOCK;
  MVAL(TOPC1) = FOR\BLOCK -> GOTO RETFOR;
  GOTO DELPTH;

```

```

EVDEPENV:
  NAME\INDEX <- FIND\NAME(NS, NP, NSV1);
  NAME\INDEX # 0 ->
    BEGIN
      EVRES <- NS[NAME\INDEX].VALUE;
      RETURN
    END;
  P <- PATH;

```

```

EVDEPENV1:
  P.DS = NIL ->
    BEGIN
      EVRES <- NSV1.TLB;
      NT Return top level binding;
      RETURN
    END;
  NAME\INDEX <- FIND\NAME(P.DS.NS, P.DSN, NSV1);
  NAME\INDEX # 0 ->
    BEGIN
      EVRES <- P.DS.NS[NAME\INDEX].VALUE;
      RETURN
    END;
  P <- P.DS;
  GOTO EVDEPENV1;

```

Auxiliary Functions

```

ADD\DEPLIST <-
  EXPR(SON:ARPTR, PATH:ARPTR; NONE)
  BEGIN
    SON.DS <- PATH;
    PATH.LOWCP = 0 =>
      BEGIN
        NT No dependents previously;
        PATH.SPATH <- SPATH <- TRUE;
        PATH.LASTSON <- SON;
        PATH.LOWCP <- SON.DSC;
        SON.PLEV <- NIL;
        SON.LBRO <- NIL
      END;
    PATH.LOWCP = SON.DSC =>
      BEGIN
        NT New dependent at same level;
        SON.PLEV <- NIL;
        SON.LBRO <- PATH.LASTSON;
        PATH.LASTSON <- SON
      END;
    NT SON is first at lower level;
    SON.PLEV <- PATH.LASTSON;
    PATH.LASTSON <- SON;
    PATH.LOWCP <- SON.DSC
  END;

CHECK\LEV <-
  EXPR(CPLEV:INT; BOOL)
  BEGIN
    DECL P:ARPTR BYVAL PATH.LASTSON;
    DECL P1:ARPTR;
    CPLEV GE P.DSC => TRUE;
  LOOP: P1 <- CHECK\TERM(P);
    P1 = P => FALSE; NT Not all sons have terminated;
    P1 = NIL =>
      BEGIN
        PATH.LASTSON <- NIL;
        PATH.LOWCP <- 0;
        PATH.SPATH <- FALSE;
        SPATH <- FALSE;
        TRUE
      END;
    CPLEV GE P.DSC =>
      BEGIN
        PATH.LASTSON <- P1;
        PATH.LOWCP <- P1.DSC;
        TRUE
      END;
    GOTO LOOP

```

END;

```
CHECK\TERM <-
  EXPR(P:ARPTR; ARPTR)
  BEGIN
    DECL P2:ARPTR BYVAL P;
  LOOP: P2.STKEFLC => P;
    P2.LBRO = NIL => P2.PLEV;
    P2 <- P2.LBRO;
    GOTO LOOP
  END;
```

```
DDEP <-
  EXPR(FATHER:ARPTR, SON:ARPTR; ARPTR)
  BEGIN
    SON.DS = NIL => NIL;
    SON.DS = FATHER => SON;
    DDEP(FATHER, SON.DS)
  END;
  NT If SON is dependent upon FATHER, then DDEP
  returns SON if it is directly dependent
  or some path P, such that P d.d. FATHER
  and SON is dependent upon P;
```

```
FIND\CENTRY\VENTRY <-
  EXPR(N:INT, P:ARPTR; NONE)
  BEGIN
    DECL F:MODE;
    DECL R, R1:INT;
    FOR I <- CP, CP - 1, ... , 1 TILL R GT 0 DO
      BEGIN
        E <- MVAL(CS[I]);
        OR(E = FOR\BLOCK,
          E = FN\BLOCK AND CS[I].ENTERED = TRUE,
          E = ENV\BLOCK,
          E = ELOCK\BLOCK) AND CS[I].OLD\NP LT N ->
          R <- I;
        OR(E = FOR\BLOCK,
          E = FN\BLOCK,
          E = ENV\BLOCK,
          E = ELOCK\BLOCK) => R1 <- I
      END;
    P.DSC <- R;
    P.DSV <- CS[R1].OLD\VP
  END;
```

```
FIND\NENTRY <-
  EXPR(; INT)
  BEGIN
```



```

DECL R:INT;
  NT Find index of last named entry on name-stack;
  FOR I <- NP, NP - 1, ..., 1 TILL R GT 0 DO
    [ ] NS[I].NAME # NIL => R <- I [ ];
  R
END;

```

```

PREV <-
  EXPR(X:ARPTR; ARPTR)
  BEGIN
    X.LBRO # NIL => X.LBRO;
    X.PLEV # NIL => X.PLEV;
    NIL
  END;

```

```

REM\DEPLIST <-
  EXPR(SON:ARPTR, PATH:ARPTR; NONE)
  BEGIN
    DECL P1, P2:ARPTR;
    PATH.LASTSON = SON =>
      BEGIN
        (PATH.LASTSON <- PREV(SON)) # NIL =>
          PATH.LOWCP <- PATH.LASTSON.DSC;
          PATH.LOWCP <- 0
        END;
      P1 <- PATH.LASTSON;
    LOOP: (P2 <- PREV(P1)) = SON =>
      BEGIN
        P1.LBRO # NIL AND P2.PLEV # NIL =>
          [ ] P1.PLEV <- P2.PLEV; P1.LBRO <- NIL [ ];
          PREV(P1) <- PREV(P2)
        END;
        P1 <- P2;
        GOTO LOOP
      END;
  END;

```

Discussion

MDEP makes the path NSV1 dependent upon the current path.

The ACTRC of a dependent path must specify which path it is directly dependent upon and the point at which it is

dependent. For a dependent path P, the DS field specifies the direct supporter path. The DSN field specifies an index in the name-stack of DS which defines the directly accessible environment of P with respect to DS. DSC is an index into the control stack of DS which specifies the control block which, when deleted, will destroy the name-stack entry corresponding to DSN, i.e. DSC is the highest level to which control may flow in DS until P terminates. DSV specifies the lowest point on the value stack of DS which is accessible to P.

The ACTRC of a supporting path must specify which paths are directly dependent upon it and the levels at which they are dependent. For a supporting path Q, LOWCP specifies the lowest point on Q's control stack at which a path was made a d.d. LASTSON specifies the last path which was made a direct dependent. All d.d. paths at a given CP level are linked through the LBRO field of their ACTRC's. The oldest d.d. at a given CP level, i.e. one with no LBRO, points through the PLEV component of its ACTRC to the d.d. at the next highest CP level.

MDEP first checks whether or not NSV1 may become directly dependent upon PATH. In particular, it checks for self-dependency and circular dependency. If NSV1 is already

*

By lowest, we mean closest to the top of the stack, i.e. most recent.

directly dependent upon PATH, then NSV1 is removed from the list of directly dependent paths by calling REM\DEPLIST.

Next, the environment of PATH is examined to determine the values of DSN, DSC and DSV for NSV1. DSN is set to the index of the most recent named entry on the name stack of PATH. CSN is set to the stack index of the control-block which added the entry corresponding to DSN to the identifier environment. DSV is set to the index of the last entry on the value-stack used by the identifiers associated with the control-block, namely, OLD\VP for the next lower control-block. If no named NS entry is found, DSN, DSC and DSV are set to reference the ENV\BLOCK at the top of the stack.

ADD\DEPLIST is called to add NSV1 to the list of paths d.d. upon PATH. If PATH was not previously a supporting path, the SPATH flag is set in PATH's ACTRC and in the environment of the evaluator. All sub-evaluators that add identifiers to the environment of the path, namely, EVELOCK, EVFOR, and APPLY, return by transferring control to the label specified by the RETURN component of the corresponding control-block. Hence, to insure that the portion of the identifier environment accessible to NSV1 is not deleted prematurely, the RETURN component of PATH's control stack entry corresponding to DSC is modified to be the evaluator label CHECK\SUPPORT which will determine whether or not all

d.d.s at this level have terminated.

CHECK\LEV returns TRUE if all paths which are directly dependent upon PATH at a control-stack entry below CPLEV have terminated. LASTSON, LOWCP, and SPATH are updated appropriately. If control can be returned safely from the control-block, i.e. if CHECK\LEV returns TRUE, then CHECK\SUPPORT transfers control to the appropriate return label based upon the mode of the control stack entry.

The value returned by MDEP is NSV1.

DEPENV obtains the most recent binding of NSV1 in the accessible environment of PATH. The identifier environment of PATH is searched first. If no binding for NSV1 is found, then the name stack of the path specified by PATH.DS (the direct supporter path) is searched starting with PATH.DSN. If no binding is found then PATH.DS.DS is searched, etc. If no binding is ever found, then the top-level binding of NSV1 is returned. Note that DEPENDV does not copy the binding of NSV1 into the RESULT\SIOT, as does PFETCH. Thus, DEPENDV may return a reference to the environment of another path.

3.6 DELETE\PATH

Definition

```
DELETE\PATH <- CSUER(PATH:ARPTR; NONE) EVDELETEPATH;
```


Example

```
CIA("DELETE\PATH",P);
```

EvaluatorEVDELETEPATH:

```

    NSV1 = PCiar ->
        ERROR("deletion");
    PATH # PCiar ->
        ERROR("CI\procedure");
    NOT TSET(NSV1.MOD) ->
        ERROR("path\mod");
    NSV1.LASTSON = NIL -> GOTO NOSONS;
    NSV1.FLGFIG <- FALSE;
    SEARCH(NSV1);
EVDEL1:
    CLEAR(NSV1.MOD);
    RETURN\NOTHING;
NOSONS:
    DELETE\ENV(NSV1);
    NSV1.FLGFIG <- FALSE;
    NSV1.DS = NIL -> GOTO EVDEL1;
    SEARCH(NSV1.DS);
    GOTO EVDEL1

```

Auxiliary FunctionsSEARCH <-

```

    EXPR(X:ARPTR; NONE)
    BEGIN
        DECL B:BOOL;
        DECL Y, Z:ARPTR BYVAL X.LASTSON;
        X.FLGFIG => NOTHING;
        NT No action if eligible for evaluation;
    BEGIN
L:      Z <- CHECK\TERM(Y);
        Z = Y => B <- TRUE;
        NT At least one son hasn't terminated;
        Z = NIL => NOTHING;
        NT No more levels;
        Y <- Z;
        NT Else set Y to first son
            at previous level;
        GOTO L
    END;
    B => NOTHING;
    NT At least one son hasn't terminated;

```

```

DELETE\ENV(X);
  NT Otherwise we can delete environment;
  X.DS = NIL => NOTHING;
  SEARCH(X.DS) NT Search father;
END;

```

```

DELETE\ENV <-
  EXPR(P:ARPTR; NONE)
  BEGIN
    P.STKEFLG <- FALSE;
    P.VS <- P.CS <- P.NS <- NIL;
    P.NP <- P.CP <- P.VP <- 0
  END;

```

Discussion

DELETE\PATH makes NSV1 ineligible for evaluation. If NSV1 has no direct dependents, then the stack-environment of NSV1 is reclaimed. Otherwise, the stack environment is reclaimed if and only if all dependents of NSV1 are ineligible for evaluation. Hence, a path may terminate but its stack environment will remain as long as necessary. If NSV1 has no direct dependents, but is itself a dependent path, then the stack environment of NSV1.DS is reclaimed if NSV1 was the last d.d. still eligible for evaluation.

The procedure SEARCH is used to determine if a path's stack environment may be deleted. If all sons of X have terminated, then the environment of X is deleted and if X.DS#NIL, SEARCH is called recursively on X.DS.

3.7 GOTO, RETFROM

Definitions

GOTO <- CSUBR(L:LABEL; NONE) EVGOTO;

RETFROM <- CSUBR(FNAME:SYMBOL, VAL:ANY; NONE) EVRETFROM;

Examples

GOTO FCO;

RETFROM("RESUME",N);

Evaluators

```

EVGOTO: NSV1.PATH # PATH ->
          ERROR("illegal\GOTO");
SPATH AND NOT CHECK\LEV(NSV1.CPINDEX) ->
          ERROR("non\support");
IFLG -> CINTRPT(NSV1.CPINDEX);
CS[NSV1.CPINDEX].STATEMENT\LIST <- NSV1.ST\LIST;
FLUSH(CS,NSV1.CPINDEX);
CP <- NSV1.CPINDEX;
NP <- TOPC1.CUR\NP;
FLUSH(VS, TOPC1.CUR\VP);
VP <- TOPC1.CUR\VP;
GOTO EVBLK1;

```

```

EVRETFROM:
  N <- 0;
  FOR I <- CP, CP-1, ..., 1 TILL N GT 0 DO
    BEGIN
      AND(MVAL(CS[I]) = FN\BLOCK,
          CS[I].NAME = NSV2,
          CS[I].ENTERED) => N <- I;
    END;
  N = 0 -> ERROR("no\call\to\return\from");
  SPATH AND NOT CHECK\LEV(N-1) ->
    ERROR("non\support");
  IFLG -> CINTRPT(N-1);
  EVRES <- NS1;
  FLUSH(CS,N);
  CP <- N;
  GOTO PROCRET;

```

Auxiliary Function

```

CINTRPT <-
  EXPR(N:INT; NONE)
  BEGIN
    FOR I <- CP, CP-1, ..., N DO
      BEGIN
        MVAL(CS[I]) = INT\BLOCK
        AND CS[I] ["RETURN"] # "COPYRT"
          => REM\INTRPT(I)
      END;
    NO\PRO\INTS AND NO\PATH\INTS ->
      IFLG <- FALSE
  END;

```

Discussion

GOTO returns control to the statement and block specified by L. RETFROM returns control from the most recent explicit call on the procedure FNAME with VAL as result.

The actions necessary to perform a GOTO or a RETFROM are quite similar. Basically, they may be divided into three parts.

- (1) The CS index of the BLOCK\BLOCK or FN\BLOCK is found.
- (2) The path-flags SPATH and IFLG are examined to determine if the stacks may be simply flushed or if special processing is required.
- (3) The RETFROM or GOTO is performed.

The CS index of the BLOCK\BLOCK to GOTO is stored in

the CP\INDEX field of the label. The CS index of the FN\BLOCK to RETFROM is obtained by searching the control stack for the most recent FN\BLOCK with ENTERED=TRUE and whose NAME field is identical to NSV2.

We will assume, for the moment, that the special processing described in (2) is not necessary.

The GOTO is completed by storing NSV1.ST\LIST as the current statement list of the block specified by NSV1.CPINDEX, flushing the stacks to the appropriate levels, and then transferring control to EVBLK1. The RETFROM is completed by setting EVRES to be the result to be returned, flushing the control stack to the FN\BLOCK and then transferring control to PROCRET to return from the procedure.

Note that both GOTO and RETFROM flush the stacks to some higher point. Consequently, unless special checks are made, the environment required by a dependent path may be destroyed or an interrupt response will be abnormally terminated. Thus the interrupt tables will not be updated correctly. The path flags SPATH and IFLG indicate to GOTO that special processing is required before the GOTO may be completed.

*

The remaining discussion references GOTO only - the interpretation for RETFROM is essentially the same.

If SPATH is TRUE, then PATH is a supporting path. Hence, EVGOTO must determine whether or not a return to the block specified by NSV1.CPINDEX will delete part of the accessible environment of a non-terminated dependent path. CHECK\LEV is used to perform this check and update the path-dependency lists if necessary.

If IFLG is TRUE, then PATH is currently evaluating at either a path or processor interrupt level. Hence, EVGOTO must determine which interrupt responses are being 'skipped' over in returning to NSV1.CPINDEX and update the interrupt tables appropriately. CINTRPT searches the control stack for INT\BLOCKs and modifies the interrupt tables to indicate that the corresponding interrupt responses have completed.

3.8 MYPATH

Definition

```
MYPATH <- CSUBR(;ARPTR) EVMYPATH;
```

Example

```
MYPATH # PCIAR => CIA("P",X);
```

Evaluator

EVMYPATH:

```
RETURN\RESULT(PATH);
```

Discussion

MYPATH returns a pointer to the current path. Note that MYPATH is a NOFIX operator.

3.9 EVAL

Definition

```
EVAL <- CSUBR(F:FORM; ANY) EVEVAL;
```

Example

```
GOTO EVAL(TOPC1.RETURN);
```

Evaluator

```
EVEVAL: F <- NSV1;
        GOTO EVAL\FORM;
```

Discussion

EVAL evaluates the form specified by NSV1 in the current path's environment. Thus, EVAL(F) is essentially equivalent to PAP(F,MYPATH).

3.10 COPYDefinition

```
COPY <- CSUBR(P:ARPTR; ARPTR);
```

Example

```
S <- COPY(MYPATH);
```

Evaluator

```
EVCOPY: BEGIN
        NSV1 = NIL => Q <- PATH;
        Q <- NSV1
    END;

    NT Q is the path to be copied;

    Q = PATH -> SAVE\STATE(Q);
    N <- LENGTH(Q.NS)/NSQUANT;
    CALL EVGETPATH1; NT P points to the new path;
    Q # PATH AND NOT TSET(Q.MOD) ->
        ERROR("path\mod");
    NOT Q.ELCFLG -> ERROR("no\stacks");
    FOR I <- 1, ... , Q.VP DO (P.VS[I] <- Q.VS[I]);
        NT Copy value-stack;
    FOR I <- 1, ... , Q.NP DO
```



```

BEGIN
    P.NS[I].NAME <- Q.NS[I].NAME;
    P.NS[I].VALUE <-
        BEGIN
            INSTACK(Q.NS[I].VALUE, O, Q.VP, Q.VS) =>
            MAP\PTR(Q.NS[I].VALUE, Q.VS, P.VS);
            Q.NS[I].VALUE
        END
    END;
    NT MAP\PTR returns a pointer to the entry in P.VS
    corresponding to the entry in Q.VS for
    Q.NS[I].VALUE;

    FOR I <- 1, ..., Q.CP DO
        BEGIN
            P.CS[I] <- Q.CS[I];
            MVAL(P.CS[I]) = INT\BLOCK AND
            P.CS[I].CLASS = "PRO" =>
                P.CS[I]["RETURN"] <- "COPYRT"
        END;

    NT Copy stack indices;

    P.NP <- Q.NP; P.VP <- Q.VP; P.CP <- Q.CP;

    P.INTINFO <- Q.INTINFO;

    NT Copy interrupt structure;

    P.DORMANT <- Q.DORMANT;
    P.TERMINATION\FORM <- Q.TERMINATION\FORM;

    NT Make dependent upon same path if any;

    Q.DS # NIL ->
        BEGIN
            Q.PLEV # NIL =>
                BEGIN
                    P.PLEV <- Q.PLEV;
                    Q.PLEV <- NIL;
                    Q.LBRO <- P
                END;
            P.LBRO <- Q.LBRO;
            Q.LBRO <- P
        END;

    P.DS <- Q.DS; P.DSC <- Q.DSC;
    P.DSN <- Q.DSN; Q.DSV <- Q.DSV;

    Q # PATH -> CLEAR(Q.MOD);
    RETURN\RESULT(P);

```

```
COPYRT: POFC1\RETURN;
```

Discussion

Only paths which are eligible for evaluation may be copied. The new path is made the direct dependent of NSV1.DS. If the path to be copied is in the midst of evaluating processor level interrupts, then the RETURN components of the corresponding INT\BLOCKs are modified so that the processor interrupt tables will not be updated incorrectly.

3.11 CIA, CONTFATH

Definitions

```
CIA <- CSUBR(FN:SYM\RTNE, ARG:ANY; REF) EVCIA;
CONTPATH <- CSUBR(P:ARPTR; ARPTR) EVCONTPATH;
```

Examples

```
CIA("P",X);
CIA(EXPR(X:ARPTR; NONE) (LASTRUN <- X), P);
LASTRUN <- CONTPATH(LASTRUN);
```

Mode

```

SYM\RTNE <- ONEOF(SYMBOL,
                  PTR(DTPR),
                  PTR(CSUBR),
                  PTR(CEXP));

```

Evaluators

```

EVCIA:  PATH = PCIAR -> ERROR("illegal\call");
EVCIA1: TSET(PCIAR.MOD) -> GOTO EVCIA2;
        CALL ALLOW\INTERRUPT;
        NT Allow for interrupt while waiting
          for CI to become available;
        GOTO EVCIA1;

```

```

EVCIA2: PATH.CIA\ARG <-
        BEGIN
            MD(NSV1).CLASS # "PTR" =>
                ALLOC(MD(NSV1) LIKE NSV1);
            NSV1
        END;
PATH.CIA\FN <- NSV2;
PUSHC("RETCI"); NT Return label for
                    when control returns;
NSV1.INPROI <-
        BEGIN
            NOT NO\PRO\INTS => PROCNUM;
            0
        END;
P.INPROI = 0 -> PRO\PATH[PROCNUM] <- NIL;
SAVE\STATE(PATH);
CLEAR(PATH.MOD);
EVRESULT(PATH, ARPTR);
INSTALL\STATE(PCIAR);
RETURN; NT Return in CI environment;

```

```

RETCI:  RETURN\RESULT(PATH.CIA\RESULT);

```

```

EVCONTPATH:
    OR(NSV1 = NIL,
        PATH # PCIAR,
        NSV1 = PCIAR,
        NOT NSV1.ELGFLG,
        NSV1.DORMANT) -> ERROR("ineligible\path");
    NOT TSET(NSV1.MOD) -> ERROR("path\mod");
    BEGIN
        NSV1.INPROI # 0 OR PRO\PATH[PROCNUM] # NIL =>

```

```

        BEGIN
            NSV1.INPROI # PROCNUM OR
                P # PRO\PATH[PROCNUM] =>
                    ERROR1("ineligible\path",NSV1);
            NSV1.INPROI <- 0
        END;
        PRO\PATH[PROCNUM] <- NSV1
    END;
    SAVE\STATE(PCIAR);
    CLEAR(PCIAR.MOD);
    INSTALL\STATE(PRO\PATH[PROCNUM]);
    CALL CHECK\INTERRUPT;
    RETURN; NT Return in environment of path;

```

Auxiliary Functions

```

SAVE\STATE <-
    EXPR(P:ARPTR; NONE)
    BEGIN
        P.NP <- NP;
        P.VP <- VP;
        P.CP <- CP;
        P.SPATH <- SPATH;
        P.IFLC <- IFLC
    END;

```

```

INSTALL\STATE <-
    EXPR(P:ARPTR; NONE)
    BEGIN
        NS <- P.NS;
        VS <- P.VS;
        CS <- P.CS;
        PATH <- P;
        NP <- P.NP;
        VP <- P.VP;
        CP <- P.CP;
        SPATH <- P.SPATH;
        IFLC <- P.IFLC
    END;

```


Discussion

CIA and CONTPATH switch control to and from the CI path, respectively.

The first argument to CIA, NSV2, is either the name of a procedure to be applied in the CI environment or a pointer to the procedure itself. NSV1 is to be the argument to the procedure specified by NSV2. If NSV1 is not of mode class PTR, then it is copied into the heap and the argument to NSV2 is the pointer to the copy.

Before control can be switched from the path to the CI, the evaluator must be sure that no other one is evaluating the CI path. Hence, it performs a busy wait on the field PCIAR.MOD. When another evaluator switches control out of the CI, it clears PCIAR.MOD. Thus, TSET will eventually return TRUE and the CIA may proceed. Note that once the TSET returns TRUE, all other evaluators that attempt CIA calls will be forced into busy waits.

The INPROI field of a path's ACTRC and the global vector PRO\PATH are used to insure that if a path performs a CIA call while processing a processor level interrupt, then the processor will not be given to another path nor will the path be evaluated by a different processor, c.f 2.5.3.

When evaluator I is evaluating path P, then PRO\PATH[I]=P and P.INPROI=0.

When evaluator I is evaluating the CI path due to a CIA call made by path P not during a processor level interrupt then $PRO\backslash PATH[I]=NIL$ and $P.INPROI=0$.

When evaluator I is evaluating the CI path due to a CIA call made by P during a processor level interrupt then $PRO\backslash PATH[I]=P$ and $P.INPROI=I$.

Having set $INPROI$ and $PRO\backslash PATH$ appropriately, the 'state' of $PATH$ is saved and its MOD field is cleared since it is not active or being modified. Hence, it may be PAPed into while the CIA call is being evaluated. $EVRES$ is set to the result to be returned in the CI environment, namely $PATH$. The 'state' of the CI is restored and a $RETURN$ is made in the CI environment. Since control can only leave the CI via a call to $CONTPATH$ and since it is not possible to PAP into the CI environment, the $RETURN$ will cause a return from the call to $CONTPATH$ with the $ARPTR$ of the path performing the CIA call as the result.

$CONTPATH$, having determined that control may be switched from the CI to the path specified by $NSV1$, saves the state of the CI path and clears $PCIAR.MOD$ so that a busy waiting evaluator may gain access to the CI. $NSV1$ is then installed as the current path. Before a $RETURN$ is made in the environment of $NSV1$, $CONTPATH$ checks for any pending pro-level or path-level interrupts.

If control ever returns to RETCI, then PATH.CIA\RESULT is returned as the value of the CIA call.

3.12 ENABLE\PRO, DISABLE\PRO, LEVEL, INUSE

Definitions

```
ENABLE\PRO <- CSUBR(EINAME:SYMBOL,
                   L:INT,
                   RESP:FORM; NONE) EVENABLE\PRO;
DISABLE\PRO <- CSUBR(EINAME:SYMBOL; NONE) EVDISABLE\PRO;
LEVEL <- CSUBR(EINAME:SYMBOL; INT) EVLEVEL;
INUSE <- CSUBR(LEV:INT; SYMBOL) EVINUSE;
```

Examples

```
ENABLE\PRO("PRO\PRO", 1, PRO\PRO\FORM);
DISABLE\PRO("LIGHT\PEN");
INUSE(LEVEL("TIMER")) = "TIMER";
```

Modes

```
SROW <- ONEOF(ROW(NPROLEV,SYMBOL),ROW(NPALEV,SYMBOL));
PITE <- STRUCT(WAITLEV:INT,
               CURLEV:INT,
               WAITING:ROW(NPROLEV,BOOL),
               INPROC:ROW(NPROLEV,BOOL),
               TYPE:ROW(NPROLEV,SYMBOL));
```

Global Variables

```

RESPONSE      ; NT A ROW(NPROC, ROW(NPROLEV,FORM));
PRO\INT\TAB    ; NT A ROW(NPROC,FITE);

```

Evaluators

```

EVENABLE\PRO:
    LEV(NSV3, PRO\INT\TAB[PROCNUM].TYPE) # 0 OR
    IN\USE(NSV2, PRO\INT\TAB[PROCNUM].TYPE) # NIL =>
        ERROR("pro\interrupt");
    PRO\INT\TAB[PROCNUM].TYPE[NSV2] <- NSV3;
        NT Indicate name of interrupt;
    RESPONSE[PROCNUM][NSV2] <- NSV1;
        NT Set up response form;
    ENABLE\PROCESSOR(NSV3, NSV2);
    RETURN\NOTHING;

```

```

EVDISABLE\PRO:
    L <- LEV(NSV1, PRO\INT\TAB[PROCNUM].TYPE);
    L = 0 => RETURN\NOTHING;
    PRO\INT\TAB[PROCNUM].TYPE[L] <- NIL;
    RESPONSE[PROCNUM][L] <- NIL;
    DISABLE\PROCESSOR(NSV1, L);

```

```

EVLEVEL:RETURN\RESULT(LEV(NSV1,PRO\INT\TAB[PROCNUM].TYPE));

```

```

EVINUSE:
    RETURN\RESULT(IN\USE(NSV1,PRO\INT\TAB[PROCNUM].TYPE));

```

Auxiliary Functions

```

LEV <-
    EXPR(N:SYMBOL, R:SROW; INT)
    BEGIN
        DECL L:INT;
        N = NIL => 0;
        FOR I <- 1, ..., LENGTH(R) TILL L GT 0 DO
            [ ] R[I] = N => L <- I [ ];
        L
    END;

```

```

IN\USE <-
    EXPR(N:INT, R:SROW; SYMBOL)
    [ ] N GT LENGTH(R) OR N LT 1 => 0; R[N] [ ];

```


Discussion

The global tables PRO\INT\TAB and RESPONSE describe the current state of the processor interrupt structure, where the i th entry in each table describes the state of the i th processor.

The PRO\INT\TAB is a row of PITEs. The fields of a PITE and their interpretations are as follows.

- (1) TYPE[J] is the symbolic name of the interrupt enabled at level J.
- (2) WAITING[J] is TRUE if and only if a TYPE[J] interrupt has occurred and the associated response form has not yet been evaluated.
- (3) INPROG[J] is TRUE if and only if the evaluation of the response form for a TYPE[J] interrupt has been initiated but has not yet completed, i.e. the response is in progress.
- (4) WAITLEV - the level of the highest priority waiting interrupt, or NPROLEV+1 if no interrupts are waiting.
- (5) CURLEV - the highest priority level of the response forms currently in progress, or NPROLEV+1 if no response forms are in progress.

RESPONSE[I][J] is the response form associated with the interrupt enabled at level J on processor I.

ENABLE\PRO enables the current processor for EENAME interrupts at level L with response form RESP, if LEV is not already in use and the processor is not already enabled for EENAME interrupts at some level. ENABLE\PROCESSOR performs any machine-dependent actions necessary to enable the processor.

DISABLE\PRO disables the current processor with respect to EENAME interrupts by setting the appropriate entries in the PRO\INT\TAB and RESPONSE tables to NIL and calling upon DISABLE\PROCESSOR to perform any necessary machine-dependent actions.

LEVEL returns the level at which the processor is enabled for EENAME interrupts, or 0 if it is not enabled.

INUSE returns the symbolic name of the interrupt enabled at level N, or NIL if LEV is out of bounds or the processor is not enabled at that level.

3.13 ENABLE\PATH, DISABLE\PATH

Definitions

```
ENABLE\PATH <- CSUBR(PEENAME:SYMBOL,
                     EV:INT,
                     ESP:FORM,
                     PATH:ARPTR; NONE) EVENABLE\PATH;
```

```
DISABLE\PATH <- CSUBR(PEINAME:SYMBOL,
                      PATH:ARPTR; NONE) EVDISABLE\PATH;
```

Examples

```
ENABLE\PATH("CI\TO\PATH",1,CI\PATH\FORM);
DISABLE\PATH("WALDO", P);
```

Mode

```
ITE <- STRUCT(WAITLEV:INT,
              CURLEV:INT,
              WAITING:ROW(NPALEV,BOOL),
              INPROG:ROW(NPALEV,BOOL),
              RESP:ROW(NPALEV,FORM),
              TYPE:ROW(NPALEV,SYMBOL),
              MASK:ROW(NPALEV,BOOL));
```

Evaluator

```
EVENABLE\PATH:
  P <- FIXPATH(NSV1);
  LEV(NSV4, P.INTINFO.TYPE) # 0 OR
  IN\USE(NSV3, P.INTINFO.TYPE) # NIL ->
    ERROR("path\interrupt");
  P.INTINFO.TYPE[NSV3] <- NSV4;
  NT Indicate name;
  P.INTINFO.RESP[NSV3] <- NSV2;
  NT Indicate response form;
  P # PATH -> CLEAR(P.MOD);
  RETURN\NOTHING;
```

```
EVDISABLE\PATH:
  P <- FIXPATH(NSV1);
  L <- LEV(NSV2, P.INTINFO.TYPE);
  L = 0 -> GOTO RTCLEAR;
  P.INTINFO.TYPE[L] <- NIL;
  NT Clear name;
  P.INTINFO.RESP[L] <- NIL;
  NT Clear response form;
```

```
RTCLEAR: P # PATH -> CLEAR(P.MOD);
RETURN\NOTHING;
```

Auxiliary Function

```

FIXPATH <-
  EXPR(P:ARPTR; ARPTR)
  BEGIN
    P = NIL => PATH;
    P = PATH => PATH;
    NOT TSET(P.MOD) -> ERROR("path\mod");
  P
  END;

```

Discussion

The INTINFO field of a path's ACTRC describes the current state of the path's interrupt structure. The components of an ITE, the mode of INTINFO, have interpretations analogous to their counterparts in a PITE, as described in the previous section. The two additional fields are interpreted as follows:

- (1) RESP[I] is the response form associated with the pseudo interrupt enabled at level I.
- (2) MASK[I] is TRUE if and only if the path is masked against interrupts of type TYPE[I].

ENABLE\PATH enables the path specified by NSV1^{*} for PEINAME interrupts at level LEV with response form RESP unless LEV is already in use or the path is already enabled for PEINAME interrupts at some level.

*
If NSV1 is NIL, then the current path is used as a default.

DISABLE\PATH disables the path specified by NSV1 with respect to PEINAME interrupts.

3.14 MASK, UNMASK, INTERRUPT

Definitions

```
MASK <- CSUBR(PEINAME:SYMBOL, PATH:ARPTR; NONE) EVMASK;
UNMASK <- CSUBR(PEINAME:SYMBOL, PATH:ARPTR; NONE) EVUNMASK;
INTERRUPT <- CSUBR(PEINAME:SYMBOL,
                   PATH:ARPTR; NONE) EVINTERRUPT;
```

Examples

```
MASK("LIGHT\PEN");
UNMASK("WALDO",P);
INTERRUPT("WALDO",P);
```

Evaluators

```
EVMASK:  P <- FIXPATH(NSV1);
          (L <- LEV(NSV2, P.INTINFO.TYPE)) = 0 ->
            GOTO RTCLEAR;
          P.INTINFO.MASK[L] <- TRUE;
          P # PATH -> CLEAR(P.MOD);
          RETURN\NOTHING;

EVUNMASK:
          P <- FIXPATH(NSV1);
          (L <- LEV(NSV2, P.INTINFO.TYPE)) = 0 ->
            GOTO RTCLEAR;
          P.INTINFO.MASK[L] <- FALSE;
          P.INTINFO.WAITLEV <-
            MINLEV\M(P.INTINFO.WAITING, P.INTINFO.MASK);
          NOT(P = PATH AND P.INTINFO.CURLEV GT
```

```

        P.INTINFO.WAITLEV) ->
            GOTO RTCLEAR;
    CALL CHECK\INTERRUPT;
    RETURN\NOTHING;

```

EVINTERRUPT:

```

    P <- FIXPATH(NSV1);
    (L <- LEV(NSV2, P.INTINFO.TYPE)) = 0 ->
        GOTO RTCLEAR;
    P.INTINFO.WAITING[L] <- TRUE;
    P.INTINFO.WAITLEV <-
        MINLEV\M(P.INTINFO.WAITING, P.INTINFO.MASK);
    NOT(P = PATH AND
        P.INTINFO.CURLEV GT P.INTINFO.WAITLEV) ->
        GOTO RTCLEAR;
    CALL CHECK\INTERRUPT;
    RETURN\NOTHING;

```

Discussion

MASK masks a path against PEINAME interrupts by setting the appropriate bit in the INTINFO field of the path's ACTRC. A PEINAME interrupt sent to the path will be detected, i.e. an entry will be made in the WAITING vector, but the response form will not be evaluated until the interrupt is unmasked.

When an interrupt is UNMASKed, the corresponding MASK bit is set to FALSE. WAITLEV is recomputed since the unmasked interrupt may have occurred, while masked, and it may be of higher priority than any of the other waiting interrupts. If the path specified by NSV1 is the current path and if WAITLEV specifies a higher priority than CURLEV, then a call to CHECK\INTERRUPT is made to initiate the

response to the interrupt at level WAITLEV.

INTERRUPT sends a PEINAME interrupt to the path specified by NSV1. An entry is made in the WAITING vector to indicate that the interrupt has occurred and WAITLEV is recomputed in case the interrupt is of higher priority than any of the other waiting interrupts. As with UNMASK, if NSV1 specifies the current path, then WAITLEV is compared with CURLEV to determine if the interrupt response should be initiated now. If the path specified by NSV1 is not the current path, then it must not be active. The interrupt response will be evaluated the next time the path becomes active, c.f. CONTPATH.

3.15 STOP\PATH

Definition

```
STOP\PATH <- CSUBR(PATH:ARPTR; NONE) EVSTOP\PATH;
```

Example

```
STOP\PATH(PAVECT[I].IDLEPATH);
```

Global Variables

```
PIL      ; NT A ROW(NPROLEV,INT);
PIV      ; NT A ROW(NPROLEV,BOOL);
```

Modes

```

INT\BLOCK <- STRUCT(TYPE:SYMBOL, INDEX:INT, RETURN:SYMBOL);
BROW <- ONEOF(ROW(NPROLEV,BOOL), ROW(NPALEV,BOOL));
FCRW <- ONEOF(ROW(NPROLEV,FORM),ROW(NPALEV,FORM));
INT\TAB\ELT <- ONEOF(ITE,PITE);

```

Evaluator

```

EVSTOP\PATH:
    NSV1 = NIL OR NSV1 = PCIAR ->
        ERROR("pro\interrupt");
    PATH # PCIAR -> ERROR("CI\procedure");
    N <- 0;
    FOR I <- 1, ... , NPROC TILL N GT 0 DO
        BEGIN
            I # PROCNUM AND PRO\PATH[I] = NSV1 => N <- I
        END;
    N = 0 -> ERROR("pro\interrupt");
    GENERATE\INT("PRO\PRO", PRO\INT\TAB[N], N);
    RETURN\NOTHING;

```

NT ALLOW\INTERRUPT determines whether a processor level interrupt has occurred.

```

ALLOW\INTERRUPT:
    NOT PIF[PROCNUM] -> RETURN;
CINT1:  NOT TSET(PIL[PROCNUM]) -> GOTO CINT1;
        PIF[PROCNUM] <- FALSE;
        IFLG <- TRUE;
        GET\INT(PRO\INT\TAB[PROCNUM],
                RESPONSE[PROCNUM],"PRO");
        NT In this case, a higher priority
           interrupt will always be found;
CINT2:  CLEAR(PIL[PROCNUM]);
        GOTO EVAL\FORM; NT GET\INT binds F to
                        the response form;

```

NT RETINT is the return label of INT\BLOCKs;


```

RETINT: NOT TSET(PIL[PROCNUM]) -> GOTO RETINT;
PIF[PROCNUM] <- FALSE;
REM\INTRPT(CP);
POPC1;

```

NT Now check for more processor
or path level interrupts;

```

MORE\INT:
BEGIN
  GET\INT(PRO\INT\TAB[PROCNUM],
    RESPONSE[PROCNUM], "PRO") =>
    IFLG <- TRUE;
  NO\PRO\INTS AND
    GET\INT(PATH.INTINFO,
      PATH.INTINFO.RESP, "PATH") =>
    IFLG <- TRUE;
  FALSE
END -> GOTO CINT2; NT Eval interrupt response;

NO\PRO\INTS AND NO\PATH\INTS -> IFLG <- FALSE;
CLEAR(PIL[PROCNUM]);
RETURN;

```

NT CHECK\INTERRUPT allows higher level
waiting interrupts to be processed, if
any exist;

```

CHECK\INTERRUPT:
  NOT TSET(PIL[PROCNUM]) -> GOTO CHECK\INTERRUPT;
  PIF[PROCNUM] <- FALSE;
  GOTO MORE\INT;

```

Auxiliary Functions

```

GENERATE\INT <-
  EXPR(EINAME:SYMBOL, TABLE:INT\TAB\ELT, N:INT; NONE)
  BEGIN
    DECL L:INT;
  LP:  NOT TSET(PIL[N]) -> GOTO LP;
    (L <- LEV(EINAME, TABLE.TYPE)) = 0 =>
      CLEAR(PIL[N]);
    TABLE.WAITING[L] OR TABLE.INPROC[L] =>
      CLEAR(PIL[N]);
    TABLE.WAITING[L] <- TRUE;
    TABLE.WAITLEV <- MINLEV(TABLE.WAITING);
    TABLE.CURLEV LE L => NOTHING;
    PIF[N] <- TRUE;
    NT The flag is set only if L is the highest
      priority interrupt which has occurred;
  
```

```

        CLEAR(PIL[N])
    END;

    REM\INTRPT <-
    EXPR(CP:INT; NONE)
    BEGIN
        DECL TABLE:INT\TAB\ELT BYREF
        BEGIN
            CS[CP].TYPE = "PRO" => PRO\INT\TAB[PROCNUM];
            PATH.INTINFO
        END;
        TABLE.INPROC[TABLE.CURLEV] <- FALSE;
        TABLE.CURLEV <- MINLEV(TABLE.INPROC)
    END;

    GET\INT <-
    EXPR(TABLE:INT\TAB\ELT, RESP:FROW, CLASS:SYMBOL; BOOL)
    BEGIN
        OR(TABLE.WAITLEV GT LENGTH(RESP),
        BEGIN
            CLASS = "PRO" => FALSE;
            TABLE.MASK[TABLE.WAITLEV]
        END,
        TABLE.WAITLEV GE TABLE.CURLEV) => FALSE;
        TABLE.INPROC[TABLE.WAITLEV] <- TRUE;
        TABLE.CURLEV <- TABLE.WAITLEV;
        TABLE.WAITING[TABLE.WAITLEV] <- FALSE;
        TABLE.WAITLEV <-
        BEGIN
            CLASS = "PRO" => MINLEV(TABLE.WAITING);
            MINLEV\M(TABLE.WAITING, TABLE.MASK)
        END;
        PUSHC(CONST(INT\BLOCK OF CLASS,
            TABLE.CURLEV, "RETINT"));
        F <- RESP[TABLE.CURLEV];
        TRUE
    END;

```

Discussion

STOP\PATH sends the external interrupt "PRO\PRO" to the processor evaluating NSV1. The processor number is obtained from the PRO\PATH vector. For I#PROCNUM, PRO\PATH[J] is the

ARPTR of the path being evaluated by processor I, c.f. CONTIPATH.

GENERATE\INT sends an EENAME interrupt to processor N. The interrupt is 'sent' as follows.

- (a) PRO\INT\TAB[N].WAITING[J] is set to TRUE, where J is the level at which processor N is enabled for EENAME interrupts.
- (b) WAITLEV is recomputed.
- (c) If the priority of CURLEV is greater than or equal to that of WAITLEV, then no further action is necessary, since the interrupt will be processed according to its priority.
- (d) If WAITLEV is of a higher priority than CURLEV, then PIF[N] is set to TRUE in order to 'signal' the fact that a higher priority interrupt has occurred.

Note that GENERATE\INT also specifies the actions that must be taken by an external processor in order to interrupt a processor. For example, a timer interrupt may be considered as an external processor that executes

GENERATE\INT ("TIMER", PRO\INT\TAB[N],N)
after some interval of time has elapsed.

An evaluator detects that an external interrupt has occurred by CALLING ALLOW\INTERRUPT at selected points, namely before the evaluation of

- (a) the body of a procedure,

(b) the body of an iteration statement,

(c) each statement in a block.

ALLOW\INTERRUPT returns immediately if PIF[PROCNUM] is FALSE, otherwise it obtains the response form associated with the interrupt and evaluates it.

To insure that ALLOW\INTERRUPT and GENERATE\INT can both examine the PRO\INT\TAB entry without interference from the other, the global vector PIL (processor-interrupt-lock) is used to provide synchronization.

GET\INT updates PRO\INT\TAB[PROCNUM] and binds F to the appropriate response form. An INT\ELOCK is pushed on the control stack which specifies the class of interrupt (either "PRO" or "PATH"), the interrupt level and a return label ("RETINT"). GET\INT returns FALSE if no interrupt response is to be evaluated.

Upon completion of the evaluation of the response form, control is passed to RETINT. PRO\INT\TAB[PROCNUM] or PATH.INTINFO is updated as the interrupt was at processor-level or path-level, respectively. The INT\BLOCK is popped off of the stack. At this point (MORE\INT), the evaluator must determine if the priority of the highest priority waiting interrupt is greater than the priority of the interrupt associated with the response form currently in progress (i.e. the priority of the interrupt which was interrupted by the one just completed.) If so, the response

form for the highest priority waiting interrupt is evaluated, otherwise the evaluation of the previous response form is allowed to continue. Note that the path interrupt levels are of lower priority than the processor interrupt levels.

4. AUXILIARY PROCEDURES

```

ALTERN <-
  EXPR(M1:MODE, M2:MODE; BOOL)
  BEGIN
    DECL E:BOOL;
    M1 = NONE => TRUE;
    MVAL(M2.D) = EDB AND M1 = M2.D => TRUE;
    FOR I <- 1, ... , LENGTH(VAL(M2, D)) TILL B DO
      [ ] M2.D[I] = M1 => B <- TRUE (];
    B
  END;

```

```

CADR <- EXPR(F:FORM; FORM) F.CDR.CAR;

```

```

CADDR <- EXPR(F:FORM; FORM) F.CDR.CDR.CAR;

```

```

CADDRR <- EXPR(F:FORM; FORM) F.CDR.CDR.CDR.CAR;

```

```

CHECKM <-
  EXPR(M:MODE; BOOL)
  [ ] MVAL(EVRES) # M => ERROR("type\fault"); TRUE (];

```

```

COMPATIBLE <-
  EXPR(SINK:MODE, SOBJ:REF; BOOL)
  BEGIN
    DECL SOURCE:MODE BYVAL MVAL(SOBJ);
    SINK = SOURCE => TRUE;
    SINK.CLASS = "PTR" AND SOURCE = NONE => TRUE;
    SINK = REF AND SOURCE.CLASS = "PTR" => TRUE;
    NOT (SINK.CLASS = "PTR" AND
          SOURCE.CLASS = "PTR") =>
      FALSE;
    ALTERN(MVAL(VAL(SOBJ)), SINK) => TRUE;
    FALSE
  END;

```

```

DEREF <-
  EXPR(R:REF; NONE)
  [ ] MVAL(R).CLASS = "PTR" => DEREF(EVRES <- VAL(R)) (];

```

```

ERROR <-
  EXPR(S:SYMBOL; NONE)
  BEGIN
    DECL N:INT BYVAL FIND\NAME(NS,NP,S);
    DECL EVRES:REF BYVAL [ ] N = 0 => S.TLB;
                                NS[N].VALUE [ ];
    DEREf(EVRES);
    NOT OR(MVAL{EVRES} = CEXPR,
           MVAL{EVRES} = CSUBR,
           MVAL{EVRES} = DTPR
           AND EVRES.CAR = "EXPR!") =>
      BEGIN
        PRINT("ERROR");
        PRINT(S);
        ERROR2()
      END;
    F <- CONS(ALLOC(REF LIKE EVRES) NIL);
    GOTO EVAL\FORM
  END;

```

NT If S is bound in the current environment to a procedure definition, then the procedure is evaluated. Otherwise, an error message is printed and ERROR2 is called;

```

ERROR1 <-
  EXPR(S:SYMBOL, P:ARPTR; NONE)
  BEGIN
    CLEAR(P.MOD);
    ERROR(S)
  END;

```

NT ERROR1 is called whenever an error occurs and the MOD field of a path has been set by a control primitive. ERROR1 clears the MOD field and calls ERROR.

```

GENV <-
  EXPR(M:MODE, P:ARPTR; REF)
  BEGIN
    DFCL VS:VSPTR BYREF [ ] P = NIL => VS; P.VS [ ];
    DECL VP:INT BYREF [ ] P = NIL => VP; P.VP [ ];
    VP <- VP + 1;
    PUSH(CONST(M),VS)
  END;

```

```

MINLEV\M <-
  EXPR(TABLE:BROW, MASK:BROW; INT)
  BEGIN
    DECL L:INT;
    FOR I <- 1, ... , LENGTH(TABLE) TILL I GT 0 DO

```

```

      [ ] TABLE[I] AND NOT MASK[I] => L <- I [ ];
      L = 0 => LENGTH(TABLE) + 1;
      L
END;

```

```

MINLEV <-
  EXPR(TABLE:BROW; INT)
  BEGIN
    DECL L:INT;
    FOR I <- 1, ..., LENGTH(TABLE) TILL L GT 0 DO
      [ ] TABLE[I] => L <- I [ ];
      L = 0 => LENGTH(TABLE) + 1;
      L
    END;
  END;

```

```

MVAL <- EXPR(P:REF; MODE) (MD(VAL(P)));

```

```

POPC <- EXPR(N:INT;NONE) [ ] FLUSH(CS,CP-N); CP <- CP-N [ ];

```

```

PURE\VALUE <-
  EXPR(; BOOL)
  BEGIN
    RSP = 0 => FALSE;
    EVRES = RESULT\SLOT[RSP]
  END;

```

```

PUSHC <-
  EXPR(CELT:ANY, P:ARPTR; NONE)
  BEGIN
    DECL CS:CSPTR BYREF [ ] P = NIL => CS; P.CS [ ];
    DECL CP:INT BYREF [ ] P = NIL => CP; P.CP [ ];
    CP <- CP + 1;
    MD(CELT) =REF => PUSH(VAL(CELT),CS);
    PUSH(CELT,CS)
  END;

```

```

PUSHN <-
  EXPR(NAME:SYMBOL, V:REF, P:ARPTR; NONE)
  BEGIN
    DECL NS:NSPTR BYREF [ ] P = NIL => NS; P.NS [ ];
    DECL NP:INT BYREF [ ] P = NIL => NP; P.NP [ ];
    NP <- NP + 1;
    NS[NP].NAME <- NAME;
    NS[NP].VALUE <- V
  END;

```



```

PUSHR <-
  EXPR(PRES:ANY, RESMODE:MODE; REF)
  BEGIN
    BEGIN
      RSP <- RSP + 1;
      RESMODE.CLASS = "PTR" AND MD(PRES) = REF =>
        BEGIN
          PUSH(CONST(RESMODE),RESULT\SLOT);
          ASSIGN(RESULT\SLOT[RSP],PRES)
        END;
      RESMODE # BEGIN
        MD(PRES) = REF => MVAL(PRES);
        MD(PRES)
      END =>
        ERROR("type\fault");
      MD(PRES) = REF => PUSH(VAL(PRES),RESULT\SLOT);
      PUSH(PRES,RESULT\SLOT)
    END;
  RESULT\SLOT[RSP]
END;

```

```

PUSHV <-
  EXPR(V:ANY, P:ARPTR; REF)
  BEGIN
    DECL VS:VSPTR BYREF [] P = NIL => VS; P.VS ([]);
    DECL VP:INT BYREF [] P = NIL => VP; P.VP ([]);
    VP <- VP + 1;
    MD(V) = REF => PUSH(VAL(V),VS);
    PUSH(V,VS)
  END;

```

```

RETURN\RESULT <-
  EXPR(X:ANY; NONE)
  BEGIN
    EVRESULT(X, MD(X));
    RETURN
  END;

```

NOFIX Operators

```
NO\PATH\INTS <- EXPR(;BOOL)
                (PATH.INTINFO.CURLEV GT NPALEV);
```

```
NO\PRO\INTS <- EXPR(;BOOL)
                (PRO\INT\TAB[PROCNUM].CURLEV GT NPROLEV);
```

```
NS1 <- EXPR(;REF) NS[NP];
```

```
NS2 <- EXPR(;REF) NS[NP-1];
```

```
NS3 <- EXPR(;REF) NS[NP-2];
```

```
NS4 <- EXPR(;REF) NS[NP-3];
```

```
NS5 <- EXPR(;REF) NS[NP-4];
```

```
NSV1 <- EXPR(;ANY) (VAL(NS1));
```

```
NSV2 <- EXPR(;ANY) (VAL(NS2));
```

```
NSV3 <- EXPR(;ANY) (VAL(NS3));
```

```
NSV4 <- EXPR(;ANY) (VAL(NS4));
```

```
NSV5 <- EXPR(;ANY) (VAL(NS5));
```

```
POPC1 <- EXPR(;ANY)
  BEGIN
    DECL R:ANY BYVAL VAL(CS[CP]);
    FLUSH(CS,CP-1);
    CP <- CP - 1;
    R
  END;
```

```
POPC1\RETURN <- EXPR(;NONE) [ ] POPC1 ; RETURN [ ];
```

```
RETURN <- EXPR(;NONE)
  BEGIN
    DECL S:SYMBOL;
    MVAL(TOPC1) # SYMBOL => GOTO EVAL(TOPC1["RETURN"]);
    S <- POPC1;
    GOTO EVAL(S)
  END;
```

```
RETURN\NOTHING <- EXPR(;NONE) (RETURN\RESULT(NIL));
```

```
TOPC1 <- EXPR(;REF) CS[CP];
```

```
TOPC2 <- EXPR(;REF) CS[CP-1];
```

```
TOPC3 <- EXPR(;REF) CS[CP-2];
```

```
TOPC4 <- EXPR(;REF) CS[CP-3];
```

```
TOPC5 <- EXPR(;REF) CS[CP-4];
```

5. PRIMITIVE PROCEDURES

A procedure is a linguistic primitive if it is used by the evaluator but it is not defined therein. It is assumed to be primitive for one of the following reasons:

- (1) Its definition is elementary and conforms to standard usage, e.g. integer addition.
- (2) It represents a language construct whose definition has no interaction with the control subroutines, e.g. the mode constructors, and for which an adequate definition is given in [Weg70].
- (3) Its definition involves machine-dependent concepts, e.g. a test-and-set instruction.

For each procedure, the arguments and result-type are given in the format of a code-procedure heading and the definition is given in English.

Arithmetic Operations

`+, -, *, /` \leftarrow CEXPR($X:INT$, $Y:INT$; INT);

Integer addition, subtraction, multiplication and division are defined with the customary interpretations.

Relational Operations - Arithmetic

`LT, LE, GT, GE` \leftarrow CEXPR($X:INT$, $Y:INT$; $BOOL$);

Returns TRUE if and only if X is less than, less than or equal to, greater than, or greater than or equal to y, respectively.

Relational Operations - General

```
=,# <- CEXPR(X:ONEOF(INT,BOOL,CHAR,REF),
              Y:ONEOF(INT,BOOL,CHAR,REF);BOOL)
```

Returns TRUE if and only if X and Y are of the same mode and are identical. For REFs, X and Y must point to the same object. #'(a,b) returns TRUE if and only if =(a,b) returns FALSE.

Logical Operations

```
NOT <- CEXPR(X:BOOL; BOOL);
```

Returns TRUE if and only if X is FALSE. NOT is a PREFIX operator.

```
AND <- CEXPR(X:FORM LISTED; BOOL)
```

If X is NIL, then AND returns TRUE. Otherwise, if each form on the list evaluates to TRUE, then the result is TRUE. If any form evaluates to FALSE or a non-boolean value, then the result is FALSE and the remaining forms are not evaluated.

```
OR <- CEXPR(X:FORM LISTED; BOOL)
```

If X is NIL, then OR returns FALSE. Otherwise, if each form on the list evaluates to FALSE or a non-boolean value, then the result is FALSE. If any form evaluates to TRUE then the result is TRUE and the remaining forms are not evaluated.

*

Mode Constructing Operations

```
ROW <- CEXPR(X:FORM LISTED; MODE)
```

X must be a list of the form

```
(id f mform)
```

or

```
(id NIL mform)
```

In the former case, f must evaluate to an integer and the mode created is 'row of eval(f) eval(mform)s'. In the latter case, the mode created is 'length unresolved row of eval(mform)s'. The CLASS field of the DDB created is "ROW".

*

For each of the mode constructors, X.CAR is either NIL or an identifier which is to be the 'shortname' of the mode created. Shortnames are used for forward references in mode definition, i.e. the shortname of a mode which has not yet been created may be used in a mode definition. For example, the actual definition of DTPR and FORM are:

```
DTPR <- STRUCT(CAR:"FORM", CDR:"FORM");
```

```
FORM <- FORM::PTR(INT, BOOL, CHAR, DDB, DTPR);
```

In the definition of DTPR, the mode FORM is referenced by its symbolic shortname. 'FORM::' specifies that the shortname of the mode produced is to be "FORM". For a complete discussion of forward reference in EL1 mode definitions see [Weg71].

```
STRUCT <- CEXPR(X:FORM LISTED ; MODE)
```

X must be a list of the form

```
( id (id-1 mform-1) . . . (id-n mform-n))
```

The mode 'structure whose i th component is of mode eval(mform-i) and has selector id-i' is created. The CLASS field of the DDB created is "STRUCT" and the D field points to a ROW(STRUCT(SYM:SYMBOL, TYPE:MODE)) of length n, where D[i].SYM=id-i and D[i].TYPE=eval(mform-i).

```
PTR <- CEXPR(X:FORM LISTED: MODE)
```

X must be a list of the form

```
( id mform-1 ... mform-n)
```

The mode 'pointer to objects of modes eval(mform-1) ,..., eval(mform-n)' is created. The CLASS component of the created DDB is "PTR" and the D field is either the mode eval(mform-1) if n=1, or a PTR to a ROW(MODE) where D[I] = eval(mform-i).

```
ONEOF <- CEXPR(X:FORM LISTED, MODE)
```

X must be a list of the form

```
( id mform-1 ... mform-n)
```

The mode 'one of the modes eval(mform-1),...,eval(mform-n)' is created. The CLASS field of the DDE created is "GENERIC" and the D field is a pointer to a ROW(MODE), where

$D[i] = \text{eval}(\text{mform}-i)$. The primitive mode ANY is defined as ONEOF ('any-mode').

Data Object Operations

$MD \leftarrow \text{CEXP}R(X:\text{ANY}; \text{MODE})$

MD returns the mode of the object X.

$VAL \leftarrow \text{CEXP}R(X:\text{REF}; \text{ANY});$

VAL returns the object pointed to by X.

$\text{CONST} \leftarrow \text{CEXP}R(X:\text{FORM LISTED}; \text{ANY})$

$\text{ALLOC} \leftarrow \text{CEXP}R(X:\text{FORM LISTED}; \text{REF})$

ALLOC and CONST create and initialize objects of any mode. The only difference in their interpretations is that ALLOC returns a pointer to the newly created object. The list X must be in one of the following formats, c.f. Appendix 3.

(1) (mform)

(2) (mform LIKE f)

(3) (mform SIZE f1 f2 ... fn)

(4) (mform OF f1 f2 ... fn)

In each case, mform must evaluate to a mode m. In case one,

the default object of the mode m is generated. In case two, the form f is evaluated and if the modes m and $MD(eval(f))$ are compatible* then an object of mode m is generated with value identical to $eval(f)$. In case three, the results of evaluating f_1, \dots, f_n are used to length-resolve the object of mode m to be generated, i.e. $f_1 \dots f_n$ specify the dimensions of the object. In case four, if m is a mode of the form $ROW(m_1)$, then $f_1 \dots f_n$ must evaluate to objects whose modes are compatible with m_1 . If so, a ROW, say R , of length n is created with $R[i]=eval(f_i)$. Otherwise, m must be a STRUCT mode. In this case, an object of mode m is generated, the components of which are copies of the values obtained by evaluating $f_1 \dots f_n$.

```
LENGTH <- CEXPR(X:ANY; INT)
```

`LENGTH(X)` is the number of components in X , provided that `MD(X).CLASS` is either "ROW" or "STRUCT". If `MD(X).CLASS="PTR"`, then X is dereferenced, and `LENGTH` is applied to the result. Otherwise, an error occurs.

*

Basically, two modes are compatible if they are identical, or if they are PTR modes and the sink mode (M) can point to the object $VAL(eval(f))$. See section 4.5 for the formal definition of compatibility.

Stack Operations

PUSH ← CEXPR(OBJ:ANY, S:STACK; REF);

See section 4.1.3.

FLUSH ← CEXPR(S:STACK, INDEX:INT; NONE);

See section 4.1.3.

INSTACK ← CEXPR(PTR:REF,
IND1:INT,
IND2:INT,
S:STACK; BOOL);

See section 4.1.3.

HEAP ← CEXPR(PTR:REF; BOOL);

See section 4.1.3.

MAP\PTR ← CEXPR(PTR:REF, OLDSTK:STACK,
NEWSTK:STACK; REF);

OLDSTK and NEWSTK must be component-wise identical and PTR must point into OLDSTK. The result of MAP\PTR is a pointer to the object in NEWSTK which corresponds to the object referenced by PTR in OLDSTK. MAP\PTR is used only by the control primitive COPY.

Miscellaneous

XCT ← CEXPR(X:ROW(INT); NONE)

The ROW(INT) is executed as machine code in the

environment of the path being evaluated. The code must bind its result to the evaluator variable EVRES.

```
ASSIGN <- CEXPR(LEFT:REF, RIGHT:REF; NONE)
```

MD(VAL(LEFT)) must be compatible with MD(VAL(RIGHT)). The object specified by RIGHT is copied into the object specified by LEFT and MVAL(LEFT) is set to MVAL(RIGHT).

```
SELECT <- CEXPR(OBJ:REF, INDEX:INT; REF);
```

SELECT returns a pointer to the INDEX component of the object referenced by OBJ.

```
ERROR2 <- CEXPR( ; NONE)
```

ERROR2 performs machine-dependent error processing.

```
TSET1 <- CEXPR(X:INT; BOOL)
```

TSET1 is the machine-dependent operation of testing and setting the value of a machine location in one instruction.

```
CLEAR1 <- CEXPR(X:INT; NONE)
```

CLEAR1 is the machine-dependent operation of unsetting the value of a machine location in one instruction.

```
CALL <- CEXPR(X:LABEL, Y:SYMBOL; NONE)
```

Y is implicit in the notation of the evaluator and must be the label associated with the next statement. Y is pushed onto the control stack and control is transferred to X, c.f. 4.1.2. CALL is a PREFIX operator.

```
INSTALL\GLOBAL\ENV <- CEXPR( ; NONE)
```

The initial 'top-level' environment in which paths are evaluated is installed by providing top-level bindings, i.e. objects referenced by the TLB components of ATOMS, for all of the following:

- (1) the control primitives,
- (2) all linguistic primitives defined in this section (except for those under the headings stack operations and miscellaneous),
- (3) PCIAR - a pointer to the control interpreter's ACTRC,
- (4) RESPONSE - the processor level response form matrix,
- (5) the evaluator constants NPROC, NPALEV, NPROLEV,
- (6) the procedures and forms in Appendix 3.

```
ENABLE\PROCESSOR <- CEXPR(S:SYMBOL, L:INT; NONE);
```

ENABLE\PROCESSOR performs any machine-dependent actions necessary to enable the current processor for S interrupts

at level L.

```
DISABLE\PROCESSOR <- CEXPR(S:SYMBOL, L:INT; NONE);
```

DISABLE\PROCESSOR performs any machine-dependent actions necessary to disable the current processor with respect to S interrupts at level L.

Priorities of Operators

The following procedures are defined as INFIX operators. They are listed in order of decreasing priority:

*,/

+, -

LT, GT, LE, GE, =, #

AND

OR

<- (assignment is treated as an operator
 even though its evaluation is via
 a sub-evaluator)

e.g. X <- A+B=C AND D=E OR F

is equivalent to

```
(X <- (((A+B)=C) AND (D=E)) OR F))
```

6. INDEX TO CHAPTER 4

#	101
*	100
+	100
-	100
/	100
=	101
ACTRC	16
ADD\DEPLIST	59
ALLOC	104
ALLOW\INTERRUPT	88
ALTERN	94
AND	101
ANY	103
APPLY	40
APPLY2	40
ARPTR	16
ASSIGN	107
ASSIGN\BLOCK	34
ATOM	21
BINDF	41
BINDF\BLOCK	39

BLOCK\BLOCK	25
BROW	88
CADDDR	94
CADDR	94
CADR	94
CALL	108
CEXP	39
CHECKM	94
CHECK\INTERRUPT	89
CHECK\LEV	59
CHECK\PATH	52
CHECK\SUPPORT	58
CHECK\TERM	60
CIA	74
CINTRPT	68
CI\PATH\FORM	48
CLEAR	56
CLEAR1	107
COMPATIBLE	94
COND\BLOCK	29
CONST	104
CONTPATH	74
CONTROL\STACK	16
COPY	72
COPYRT	74
CSPTR	16

CSQUANT	48
CSUBR	39
DDB	15
DDEP	60
DECL\BLOCK	27
DELETE\ENV	66
DELETE\PATH	64
DELPTH:	49
DEPENV	57
DEREF	94
DISABLE\PATH	83
DISABLE\PRO	79
DISABLE\PROCESSOR	109
DOPAP	51
DPAP	50
DPAPQ	50
DTPR	16
ENABLE \PATH	82
ENABLE\PRO	79
ENABLE\PROCESSOR	108
ENV\BLOCK	48
ERROR	95
ERROR1	95
ERROR2	107
EVAL	71

EVALUATOR	17
EVAL\FORM	21
EVASSIGN	34
EVBLK1	25
EVBLOCK	25
EVCIA	75
EVCLAUSE	30
EVCLEAR	56
EVCONTPATH	75
EVCOPY	72
EVDECL	27
EVDELETEPATH	65
EVDEPENV	58
EVDISABLE\PATH	83
EVDISABLE\PRO	80
EVD PAP	50
EVDTPR	23
EVENABLE\PATH	83
EVENABLE\PRO:	80
EVEVAL	72
EVEXPR	24
EVFOR	36
EVGETPATH	48
EVGETPATH1	48
EVGOTO	67
EVIF	30

EVINTERRUPT	86
EVINUSE	80
EVLABST	46
EVLEVEL	80
EVMASK	85
EVMDEP	57
EVMYPATH	71
EVPAF	50
EVPFETCH	55
EVSTORE	55
EVRESULT	21
EVRETFROM	67
EVSEL	31
EVSELQ	32
EVSTOP\PATH	88
EVSVM	21
EVTSET	56
EVUNMASK	85
EXISTS	52
FIND\CENTRY\VENTRY	60
FIND\NAME	22
FIND\NENTRY	60
FIXPATH	84
FLUSH	106
FN\BLOCK	39
FORM	15

FOR\BLOCK	35
FROM	88
GE	100
GENERATE\INT	89
GENV	95
GET\INT	90
GET\PATH	48
GOTO	67
GT	100
HEAP	106
IDLE	15
INIT\INTERRUPTS	18
INIT\STATE	15
INSET	52
INSTACK	106
INSTALL\GLOBAL\ENV	108
INSTALL\STATE	76
INTERRUPT	85
INT\BLOCK	88
INT\TAB\ELT	88
INUSE	79
IN\USE	80
ITE	83
LABEL	27

LE	100
LENGTH	105
LEV	80
LEVEL	79
LT	100
MAP\PTR	106
MASK	85
MD	104
MDEP	57
MINLEV	96
MINLEV\M	95
MODE	15
MOVE\ARGS	52
MVAL	96
MYPATH	70
NAME\STACK	16
NOT	101
NO\PATH\INTS	98
NO\PRO\INTS	98
NPALV	15
NPROC	15
NPROLEV	15
NSi	98
NSPTR	16
NSQUANT	48

NSVi	98
ONEOF	103
OR	102
PAP	50
PAPF	51
PAPQ	50
PAP\ELOCK	50
PCIAR	15
PFETCH	54
PIL	87
PITE	79
PIV	87
POPC	96
POPC1	98
POPC1\RETURN	98
PREV	61
PROCRET	40
PROC\EXIT	43
PRO\INT\TAB	80
PRO\PATH	15
PRO\PRO\FORM	15
PSTORE	54
PTR	103
PURE\VALUE	96
PUSH	106

PUSHC	96
PUSHN	96
PUSHR	97
PUSHV	97
PUTNAMES	42
REM\DEPLIST	61
REM\INTRPT	90
RESOLVE	42
RESPONSE	80
RETBLOCK	25
RETCI	75
RETFN	41
RETFOR	37
RETFROM	67
RETINT	89
RETURN	98
RETURN\NOTHING	99
RETURN\RESULT	97
ROW	102
RTCLEAR	83
SAVE\STATE	76
SAVE\VAL	32
SEARCH	65
SELECT	107
SELECTOR\INDEX	32

SEL\BLOCK	31
SETUP	55
SIGN	37
SROW	79
STOP\PATH	87
STRING	21
STRUCT	103
SYMBOL	21
SYM\RTNE	75
TIMER\FORM	15
TIME\OUT\FORM	48
TOPCi	99
TSET	56
TSET1	107
UNMASK	85
UNSAVE\VAL	32
VAL	104
VALUE\STACK	16
VSPTR	16
VSQUANT	48
XCT	106

Chapter 5

EVALUATION AND CONCLUSIONS

1. OTHER FACILITIES

In this section, we consider a number of features of MPEL¹ which were mentioned in chapter 2 or utilized in chapter 3, but for which no detailed explanation has yet been given.

1.1 Extended CIA Call

In section 2.3.1, we indicated that it is possible, by extension, to CIA call procedures that take more than a single argument, but deferred explanation. Here, we remedy this omission and consider one additional point.

The procedure ECIA (extended-CIA) takes an indefinite number of arguments. The first argument specifies the procedure to be applied in the CI's environment (as in CIA.) The remaining arguments to ECIA are evaluated to yield the arguments for the procedure application. ECIA constructs a list whose first element is the procedure specification and whose tail is a list of REFs that point to the evaluated

*
arguments. ECIA then performs a CIA call on the procedure EVAL with this list as argument. The list is evaluated in the environment of the CI as a call to the procedure with the specified arguments. Thus, ECIA effects the CIA call of a procedure with an arbitrary number of arguments. The definition of ECIA is as follows.

```

ECIA <-
  EXPR(FN:ONEOF(SYMBOL,ROUTINE),ARGS:FORM LISTED;REF)
  BEGIN
    DECL CIARES:REF BYREF ALLOC(REF);
    MYPATH.CIA\RESULT <- CIARES;
    CIA("EVAL", CONS(BEGIN
                      MD(FN)=SYMBOL => FN;
                      ALLOC(REF LIKE FN)
                      END,
                      EVALLIST(ARGS)));
    VAL(CIARES)
  END;

EVALLIST <- EXPR(ARGS:FORM; FORM)
  BEGIN
    ARGS=NIL => NIL;
    BEGIN
      DECL A:ANY BYREF EVAL(ARGS.CAR);
      DECL R:REF BYVAL
      BEGIN
        MD(A).CLASS="PTR" => A;
        ALLOC(MD(A) LIKE A)
      END;
      NT If not a PTR mode, allocate as in CIA;
      CONS(ALLOC(REF LIKE ALLOC(REF LIKE R)),
           EVALLIST(ARGS.CDR))
    END
  END;
END;

```

ECIA requires that the arguments to the ECIA called procedure be of mode class PTR. This is consistent with the definition of CIA. A slightly different definition of

*

In the list structure representation of MPOL1 programs, a REF evaluates to the object that it references. Hence, it acts as a QUOTE for arbitrary objects, c.f. 4.2.2.

EVALLIST allows the arguments to be of any mode, viz.

```

EVALLIST <- FXPR(ARGS:FORM; FORM)
  BEGIN
    ARGS=NIL => NIL;
    BEGIN
      DECL A:ANY BYREF EVAL(ARGS.CAR);
      DECL R:REF BYVAL ALLOC(MD(A) LIKE A);
      CONS(ALLOC(REF LIKE R), EVALLIST(ARGS.CDR))
    END
  END;

```

Since the arguments are allocated in the heap, they are, in effect, always passed BYVAL.

ECIA differs from CIA in that it does not return the REF specified by the CIA\RESULT component of the path's ACTRC. Instead, it allocates a REF in the heap, stores a pointer to the REF in the CIA\RESULT component, and after the CIA call on EVAL, returns the allocated REF as result. If a procedure which is ECIA called wishes to have a value, say R, returned as the result of the call to ECIA, then it must execute

```
VAL(LASTRUN.CIA\RESULT) <- R;
```

The indirection insures that the value will be returned correctly, even if another ECIA call is PAPed into the environment of the path. This is not the case with nested CIA calls, where it is possible for a result to be lost. For example, suppose a path, say P, CIA calls a procedure F which sets the CIA\RESULT component of P's ACTRC. If, while the CIA call is being executed, another path executes

```
PAP(CIA("F"),P)
```

then when control returns to P, the second CIA call on F

will be executed and the value stored in P.CIA\RESULT by the first call will be destroyed. The value returned by the second call will also be incorrectly returned as the result of the first call. If F is written to be ECIA called, then the values will be returned correctly since each call to ECIA retains a pointer to a distinct allocated REF which is used to hold the result.

CIA could have been defined to return its result in a manner similar to ECIA. This would require a REF to be allocated for each CIA call, whether or not the CIA called procedure returns a value. However, in our experience with CIA we have found that the majority of CIA called procedures do not return values. Thus, we have declined to include this mechanism as primitive since it is used infrequently and can be achieved by extension.

1.2 Extended Mode Facility

The examples in Chapter 3 utilize the extended mode facility of EL1 [Weg70][Weg71] in two ways. First, the MONITOR operation uses a 'user-defined' assignment function to check assignments to monitored variables on a mode-dependent basis, c.f 3.5. Second, the addition of extended components to the definition of ACTRC is achieved through the use of the facility. Here, we will discuss the

extended mode definition facility and show how it can be used to implement extended components.

Basically, the facility allows the programmer to control the behavior of a mode M by specifying EL1 procedures to be called whenever an object of mode M is to be assigned a value or whenever a component of an object of mode M is to be selected. In addition, it allows for the specification of a conversion procedure to be used in the conversion of objects of mode M to other modes as required. For example, consider the case of monitoring. Here, we would like to monitor an integer and take some action if it is assigned a certain value. Aside from assignment, the integer is to act like any other 'normal' integer. To achieve this effect, we define the mode SINT as a STRUCT with an integer component (I) that contains the monitored integer, and one or more components which hold the associated monitoring information. We then extend the mode SINT by associating^{*} with it the three functions SINT\ASSIGN, SINT\SELECT, SINT\CONVERT to be used in assignment, selection, and conversion of objects of mode SINT, respectively.

SINT\ASSIGN, discussed in section 3.5, assigns the specified value to the I component of the SINT and

*

The details as to how these functions are associated with the mode are given in [Weg70].

determines if it has been assigned the value being monitored for. Since SINT's are to act as integers they cannot be selected, even though they are structured objects. Hence, an attempt to select a component of a SINT should generate an error, viz.

```
SINT\SELECT <- EXPR(S:SINT, C:ONEOF(INT,SYMBOL); NONE)
                (SELECT\ERROR());
```

Finally, to complete the illusion that an SINT, say X, is really an integer, it is necessary to allow X to appear where an integer value is required, e.g.

X+5

To achieve this, the procedure SINT\CONVERT is called to perform the appropriate conversion whenever an SINT is in hand and an object of some other mode M is required, viz.

```
SINT\CONVERT <- EXPR(S:SINT, M:MODE; M)
BEGIN
  DECL TEMP:INT;
  M#INT => CONVERT\ERROR();
  NT Only INT conversion is defined;
  TEMP <- UR(S).I;
  NT Select the component that contains the integer;
  TEMP
END;
```

The use of UR requires some explanation. If SINT\CONVERT executes the statement

TEMP <- S.I

in order to select the I component of the SINT, then SINT\SELECT would be called to perform the selection and an error would result. Hence, it is necessary for SINT\CONVERT to specify that the selection is to be performed on the SINT

taken as an unextended mode. This is achieved by using the procedure UR which specifies that the selection is to be performed on the underlying representation of SINT. SINT\ASSIGN also uses UR in order to assign to the I component and to select the components of the SINT which contain the monitoring data.

We can now describe how the mode ACTRC can be extended to include components required by some control regime. ACTRCs contain the basic component USER\AR which is of mode REF. This component can be used to point to an object which contains the extended components. A user-defined selection function can then be used to select both basic and extended components. For example, suppose we wish to extend ACTRC to contain the four components PAL, PVALRET, PVALQ, and PAVAL used in the parallel processing examples of section 3.3. We define the mode USER\COMPS and the procedure ACTRC\SELECTION as follows.

```
USER\COMPS <- STRUCT(PAL:STRUCT(OWNER:ARPTR,WLIST:ARPTR),
                     PVALRET:BOOL
                     PVALQ:ARQPTR,
                     PAVAL:REF)
```

```

ACTRC\SELECTION <-
  EXPR(P:ACTRC, I:ONEOF(INT,SYMEOL); ANY)
  BEGIN
    DECL N:INT
    UR(P).USER\AR=NIL ->
      UR(P).USER\AR <- ALLOC(USER\COMPS);
    BEGIN
      MD(I)#INT => N<-SELECTOR\INDEX(ACTRC,I) ; *
      N <- I
    END
    N=0 => UR(P).USER\AR[I];
    N GT LENGTH(ACTRC) =>
      UR(P).USER\AR[I-LENGTH(ACTRC)];
    UR(P)[I]
    NT Select basic component;
  END;

```

When the first selection is performed on an ACTRC, a USER\COMPS is allocated and a pointer to the object is stored in USER\AR. If the component to be selected is a basic component, then selection is performed on the ACTRC. Otherwise, the appropriate component of the USER\COMPS is selected. For example, if Q is an ARPIR, then evaluation of the form Q.PVALRET will trigger the following procedure call

```
ACTRC\SELECTION(VAL(Q),"PVALRET")
```

Since PVALRET is not a basic component of the mode ACTRC, SELECTOR\INDEX(VAL(Q),"PVALRET") returns zero, and thus the PVALRET component of object referenced by Q@USER\AR is selected.

*
SELECTOR\INDEX, defined in section 4.2.8, returns the integer index associated with a symbolic selector, or zero if the symbol is not a selector of the mode.

1.3 Termination of Dependents

In section 2.2.8, we noted that it is possible to construct a procedure which will terminate all paths dependent upon the sub-environment of a path. Here, we present such a procedure, namely, TERM\DEPS.

The procedure can be called explicitly, say at the end of a block, to terminate all dependent paths created in an environment. It can also be called implicitly by binding the procedure to the identifier associated with the error condition "non\support". If an attempt is made to delete an environment accessible to a non-terminated dependent path, then TERM\DEPS is executed in response to the error condition (which results in the dependents being terminated,) and then the environment is safely deleted.

All dependents of a given path, say P, are linked together (through their ACTRCs) in a tree structure as described in section 4.3.5. P.LASTSON specifies the path most recently made directly dependent upon P. All paths with the same directly accessible environment are linked together (starting with P.LASTSON) through the LBRO component. The last of these (LBRO=NIL) is linked to the paths directly dependent upon P with 'smaller' directly accessible environments. It is only necessary to terminate, i.e. make ineligible for evaluation, all direct dependents who can reference the current directly accessible

environment and then terminate all of their dependents (recursively.) This can be achieved by performing a tree walk on the ACTRCs calling DELETE\PATH as necessary.

There is only one problem with this solution - some of the dependent paths may be active and thus, they cannot be terminated by calls to DELETE\PATH. We require a mechanism which will insure that the actions of TERM\DEFS will be relatively continuous with respect to the evaluations of all (dependent) paths. The operators STARTRC and ENDRC, described in section 3.5, provide this facility. Hence, we simply bracket the substantive portion of the procedure with these operators. TERM\DEFS and its auxiliary procedure TERM\DEPS1 are defined as follows.

```

TERM\DEFS <- EXPR(; NONE)
  BEGIN
    DECL P:ARPTR EYVAL MYPATH.LASTSON;
    STARTRC;
    NT The following block is executed
      relatively continuous to all
      other paths;
    WHILE P#NIL DO
      BEGIN
        P.ELGFLG => CIA("DELETE\PATH",P);
        TERM\DEPS1(P.LASTSON);
        NT Terminate all dependents
          of this direct dependent;
        P <- P.LBRO
        NT P is next direct dependent;
      END;
    ENDRC
    NT All dependents at current level
      have been terminated;
  END;

TERM\DEPS1 <- EXPR(P:ARPTR; NONE)
  BEGIN
    P=NIL => NOTHING;
    P.ELGFLG -> CIA("DELETE\PATH",P);
    TERM\DEPS1(P.LBRO);
  
```

```
TERM\DEPS1(P.PLEV);  
NT At most one of P.LBRO and F.PLEV  
   will be non-null;  
TERM\DEPS1(P.IASTSON)  
END;
```

2. IMPLEMENTATION ISSUES

In previous chapters we have excluded discussions of implementation issues in the interest of simplicity. Here, we will restrict ourselves to those topics which are directly related to the multi-path facility. Other issues, such as the translation of MPFL1 programs from external to internal representation, are of peripheral interest and have been adequately discussed elsewhere [Weg70].

2.1 Storage Management

In MPFL1, storage (core) is designated as being either stack or heap.

Stack storage may only be created by a call to GET\PATH, where the integer argument specifies the number of K of contiguous stack storage to be allocated for the path's environment.* Stack storage is retained until the path is explicitly deleted (via a call to DELETE\PATH,) unless the path has non-terminated dependents. In this case the storage is retained until all dependents have terminated.

Since it is not always possible to predict a priori the amount of stack storage required by an individual path, the

*

Actually, as described in chapter 4, three stacks are allocated - the name, control and value stacks.

possibility of stack overflow exists. This can be handled in one of two ways. First, the program can be aborted, with suitable error messages presented to the user. Alternatively, the stack may be automatically expanded and the path's computation resumed. The latter is obviously more desirable, since it permits a path to be created with a small stack allocation and allows for growth, as required.

A stack may be expanded either by mapping it into a larger contiguous storage region or by linking it to another stack segment.

The latter solution destroys the assumption that the stack is contiguous. This presents a number of problems. First, it degrades the efficiency of system routines that access the stack. For example, an additional check must be included in the routine that searches the name stack to determine if it is necessary to switch segments. Second, programs which run at the end of a segment will suffer the overhead of constantly switching between segments. In addition, it is difficult to impose a reasonable de-allocation policy for segments. If a segment is freed as soon as control returns to a 'higher' segment, it may be necessary to immediately re-allocate the segment if the higher one overflows again. Conversely, if the segment is not freed and the higher segment does not overflow again, then the storage is wasted. Linked segments, however, do

not require the stack to be copied as in the former solution.

The advantage of mapping the stack into a larger segment is that the implementation can assume that the stack is a contiguous block of storage. In addition, we note that the ability to copy a stack is already present due to the control primitive COPY - no additional mechanism is required. Furthermore, as we will see below, a compactifying garbage collection requires stack relocation. It is feasible to delay the execution (via scheduling) of all paths whose stacks have overflowed, and then map them all during garbage collection.

In either scheme, it is desirable that stack overflows occur at predictable points with respect to the path's evaluation, e.g. only after all formals and locals of a procedure call have been entered on the name stack. With linked segments, this at least allows the compiler to assume that all the locals and formals of a procedure are contained in a contiguous block, thereby simplifying compiled references to local variables. The mapping strategy requires that the stacks can be 'read', i.e. that the contents of each stack location can be unambiguously decoded so that those words which require relocation can be determined. Insuring that this condition holds whenever an object is pushed onto a stack during evaluation imposes

severe constraints upon the system. If overflows can only occur at certain points, then it is only necessary to insure that the condition holds at these points.

The situation described above can be effected by reserving a portion of the stack for stack extension. When a 'hard' overflow occurs, the extension is appended to the end of the stack. An interrupt is enabled which will be triggered the next time the stack is 'clean' and the path is allowed to continue. When the clean point is reached, the stack may be expanded.

Stack storage is used to hold objects whose lifetime is keyed to procedure (or block) activation, i.e. the objects are created upon procedure call and deleted upon procedure exit. Since the procedure call-block activation control structure of MPEL1 is strictly hierarchical, these objects can be managed using LIFO (last-in-first-out) stacks. MPEL1 also allows for objects whose lifetime is independent of the call structure. These objects, created by calls to the procedure ALLOC, are managed using a retention strategy, i.e. the object exists as long as it can be referenced. Storage for these objects is allocated from a free storage region called the heap. When necessary, a garbage collection is invoked to determine which storage block can no longer be referenced. These blocks may be returned to the free storage pool.

The multi-path environment of MPEL1 raises a number of issues with respect to heap management.

- (1) How is access to the free storage pool synchronized?
- (2) What is the retention strategy for paths?
- (3) Can the garbage collector make use of additional processors?

We consider each of these in turn.

*

If free blocks are contained on a single free list, then synchronization can be achieved by associating a single memory location with the list. This location is TSET by a processor before accessing the list and CLEARED once the desired free block has been removed. If the memory location has already been 'set', then the processor loops in a busy wait until the location is CLEARED.

Although the above organization is sufficient for heap management, it is not necessarily the most efficient one. For example, if parallel paths perform many allocations, then a considerable amount of each processors time may be spent in the busy wait. Thus, heap allocation may become a system bottleneck. If N free lists are used, then it is possible for N processors to obtain heap storage

*

Here we use the term 'free-list' to denote any one of a number of implementation techniques. For example, the free-list may actually be a vector of lists, where each list contains all blocks falling in a specific size range.

simultaneously. In this case, however, it may be necessary to access more than one list before a block of the desired size is found. In addition, the interlocking strategy in a multi-list scheme will be more complex than with a single global interlock. Hence, we will assume that implementations will initially adopt the latter scheme and impose more complex ones only if the need arises.

If a block of sufficiently large size cannot be found, a garbage collection is required. In single-path EL1 this involves marking all heap objects which are accessible to the program by following all pointer chains to and within the heap starting from so-called base-positions, e.g. all objects on the path's stacks and top-level bindings. After this first trace phase, unmarked heap locations correspond precisely to those objects that can no longer be referenced. At this point, two strategies are possible. Either the inaccessible blocks are simply added to the free-list or the accessible objects are mapped into one contiguous block, leaving one large free block at one end of the heap. The former strategy is called collection and the latter is called compactification.^{*} If the heap is not contiguous, i.e. there are stack segments interspersed between heap segments, then in the latter case the stacks must be mapped as well.

*

For a detailed discussion of compactifying garbage collection in the heap see [Weg71a].

In MPPEL1, the trace phase must insure that all paths that may ever become active are traced. A path is traced by tracing its activation record as a structure and considering each object on its stacks as a base position. All paths active at the time garbage collection occurs are traced. The CI is always traced. If an unmarked activation record is encountered, then it is traced as a path if and only if it still possesses a stack environment. Otherwise, only the activation record is traced (as a structure.) Since a path can only become active if its ARPTR is accessible from an active path or the CI, all potentially active paths will be traced. If, in addition, we assume that GETPATH maintains a list of the ACTRCs of all allocated paths, then this list can be scanned after the trace phase and all unmarked activation records and their associated stacks can be reclaimed since the paths can never become active.

In the discussion above, we have tacitly assumed that it is possible to stop the evaluation of other active paths so that the system may commence garbage collection. This can be accomplished as follows. When one processor wishes to invoke garbage collection, it TSETs a gc-flag location. If the gc-flag is not already set, then the processor sends a "PRO\PRO" interrupt to all other processors to indicate that they should cease evaluation. If the gc-flag is already set, then some other processor has started the garbage collection, so the processor idles waiting for the

"PRO\PRO" interrupt. In the following discussion it will be convenient to refer to the processor that successfully TSETs the gc-flag as the master processor.

The multi-processor configuration can be used to perform parts of the garbage collection in parallel. When a processor receives the "PRO\PRO" interrupt, it begins tracing the path it was evaluating. The master insures that the CI is traced, if it was not active. When an unmarked ACTRC is found, it is marked and placed upon a list of paths to be traced. When a processor completes the tracing of a path, it removes an ACTRC from the list (with suitable interlocks) and traces it as a path. If the list is empty, it idles waiting for a path to trace. When the master detects that all paths are idling (by examining an idle-flag associated with each processor,) it initiates the next phase, namely, collection or compactification. In the former case, the heap can be divided into segments and each processor assigned a segment to collect. In the latter case, after initial 'set-up' work by the master, the additional processors may be used to perform the mapping of storage in parallel. After this phase, the processors may resume evaluation.

2.2 Input\Output

Most computer systems allow input\output (I/O) activities and program execution to proceed concurrently. In this section, we will discuss how concurrent I/O can be incorporated into the framework of MPEL1. As the actual language I/O primitives are only of peripheral interest, we will simply use the generic terms READ and PRINT.

The processor level interrupt facility of MPEL1 in conjunction with an appropriate "START\IO" control primitive to communicate with the I/O processors would allow a path to handle its own concurrent I/O. However, as Wirth [Wi69] notes, it is conceptually simpler to assume that a given I/O operation is synchronous with respect to a path's evaluation, e.g. if a path performs a READ, then further evaluation of the path is delayed until the input is available. Concurrent I/O can then be realized by creating parallel paths to perform I/O operations. Synchronization can be achieved through the use of the CI framework, c.f. 3.2.

Thus, it is only necessary to show how the synchronous functions READ and PRINT can exploit a concurrent I/O facility in terms of the MPEL1 framework. Typically, a processor initiates concurrent I/O by executing an instruction which sends an interrupt to an external processor (I/O device.) The external processor indicates

that transmission has been completed by sending an interrupt back to the processor. Thus, to perform a READ, the path CIA calls a procedure which will queue the path as waiting for I/O, sets LASTRUN to NIL so that the processor will be given to another path, and then performs the "STARTIO". When the I/O-complete interrupt occurs, the path may be put on the INACTIVEQ so that it may be assigned to a processor. Equivalently, one can think of the external processor as being assigned to the path for the duration of the I/O transmission. Because the external processor cannot perform a CIA call, it is necessary to use interrupts to achieve the same effect.

2.3 Relation to an Operating System

The underlying machine model upon which MPEL1 is based can be summarized as follows:

- (1) There exist n processors available for the simultaneous evaluation of paths.^{*} A processor idles if it has no path to evaluate. The processors share a common address space.
- (2) A timer interrupt facility exists.
- (3) One processor may interrupt another via a processor-to-processor interrupt.

*

Note that the processors do not have to be identical; they may have different architectures.

- (4) A test-and-set instruction exists, which allows for processor synchronization via busy waits.

Typically, MPEL1 will be implemented in terms of a virtual machine provided by an operating system. The control structure of the virtual machine may not conform to the requirements listed above. For example, virtual processors may actually be implemented by multiplexing a single processor. Hence, they are capable of concurrent, but not simultaneous, evaluation. Here, we will discuss the implementation of MPEL1 in terms of various virtual machine organizations.

MPEL1 can be implemented on the simplest of virtual machines, namely, one which allows only one processor (a job) to access an address space. In this case, the "PRO\PRO" interrupt is not necessary since there is no other processor to communicate with. If a timer is not available, a similar effect can be achieved by counting the number of function calls made by a path and generating an interrupt after some specified number have occurred.

If the virtual machine allows many virtual processors to access a common address space, then the question arises as to how many should be used by MPEL1, i.e. what should the value of n be? Here, it is only necessary to set it to be equal to the number of real processors available, say m , as this number represents the maximum simultaneous

evaluation of which the system is capable. Of course, n can be set lower than m .

The implementation, however, cannot assume that the m virtual processors are 'real' since at any given time some of the real processors may actually be assigned to other tasks in the operating system. In particular, the concepts of 'busy-wait' and 'idling' must be re-examined. In the formal model, a processor goes into a busy wait if it attempts to transfer control to the CI and cannot do so because the CI is being evaluated by another processor. In the context of an operating system, however, the virtual processor is only a control path (with respect to the OS as CI) and hence the real processor can be re-assigned to another virtual processor. Similarly, a virtual processor can put itself to sleep instead of idling. The CI will awaken it when there exists a path to evaluate.

Both of the above can easily be achieved if the virtual machine provides a means whereby virtual processors can perform non-busy waits. For example, let us assume that semaphores and the operations p and v are available. We will associate one binary semaphore with each virtual processor ($SEMi$) and one with the CI ($CISEM$). Before passing control to the CI, a processor performs $p(CISEM)$. When a processor transfers control out of the CI it performs a $v(CISEM)$. To idle, processor i performs a $p(SEMi)$, where

the semaphore is assumed to be initially 0. Thus, the processor waits indefinitely. When the CI wishes to awaken an 'idling' processor it simply performs a v(SEM_i).

A "PRO\PRO" interrupt is still required to implement STOP\PATH(P), where P is not an idle path. Either an interrupt or a mechanism which allows one virtual processor to stop another, allows it to modify its registers, and then allows it to continue will suffice. An example of the latter mechanism appears in the TENEX [BBN70] operating system. Here, it is possible to create multiple forks (virtual processors) that access a common address space. One fork may freeze (FFORK) another, modify its state, and then allow it to continue (RFORK.)

We close this section with the observation that the multi-path facility is quite machine independent. The design allows an implementation to utilize whatever features the virtual machine provides. In addition, the parts of the system which relate to the operating system are isolated. Hence, re-implementation of the language (or at least the control facility) on another machine should be relatively simple.

3. CRITICAL DISCUSSION

In this section, we present an evaluation of the multi-path facility and its formal model. Both are examined in terms of the design criteria discussed in section 1.2. In addition, they are compared with some of the languages and models discussed in section 1.2.

In evaluating the efficiency of the primitives and framework of MPEL1, we refer frequently to their treatment in the formal specification. This is reasonable since the model is distinctly not implementation independent. The data structures used by the evaluator are essentially the ones to be used in an implementation. This is discussed in more detail below.

3.1 The Control Primitives

The problem of 'size' turns out to be largely a pseudo question. The primitives are used to define extensions for various multi-path organizations. Typically, the code for the extensions will outweigh the initial investment in the primitives. Second, in the environment of a language system, the primitives may be maintained on a library file and loaded as required. Still, one would expect the amount of code required to be small as compared with the rest of the implementation. To facilitate this, the primitives

utilize other components of the language wherever feasible. For example, the primitives are defined as control subroutines (CSUBRs), and thus their arguments may be evaluated and bound in the same fashion as EL1 procedures. The mode definition facility is used to define the data types required by the primitives. PAP uses the sub-evaluator APPLY. PFETCH, PSTORE and DEPEND all utilize the name-stack search procedure. In addition, we note that some of the auxiliary procedures required by the control primitives can be implemented quite efficiently. For example, MINLEV can be implemented in a single machine instruction if we assume reasonable values for NPALEV and NPROLEV.

We can comment upon the amount of code required for the control primitives in the current ECL implementation of MPCL1 [Weg72]. Here, only GET\PATH, DELETE\PATH, PAP, CIA, MYPATH, RETFROM, and GOTO have been implemented and constitute approximately 4% of the system code.^{*} It is expected that a complete implementation of the primitives will require roughly 1000-1500 words of code.

We turn now to the question of the effect of the

*

ECL runs on a DEC PDP-10 computer under either the TOPS or TENEX monitors. The system includes an EL1 interpreter, garbage collector, mode-definition routines, and system support code. It does not, however, include a compiler.

multi-path facility upon the evaluation of a single path of control. An examination of the EL1 evaluator presented in section 4.2 reveals four places in which the facility places additional overheads upon the EL1 evaluator. We consider each of these in turn.

Before it evaluates the body of a procedure, APPLY must determine if the body is to be evaluated in the environment of another path. This situation can arise as a result of a call to PAP. The overhead can be kept to a minimum by having PAP set a path-dependent flag. APPLY would only have to make additional checks if the flag is set.

Return from procedure calls, blocks and FOR loops is made through the RETURN component of the FN\BLOCK, BLOCK\BLOCK and FOR\BLOCK, respectively. This is required so that attempts to delete accessible environments may be trapped by simply modifying the RETURN component to be the label CHECK\SUPPORT. In the absence of path dependency, the RETURN component could be removed and the return effected implicitly by each sub-evaluator. However, the additional storage required by the inclusion of the RETURN component is negligible as compared with the total amount of stack storage required in the cases above.

A CALL to ALLOW\INTERRUPT is made before the evaluation of each statement of a block, the body of a FOR statement and the body of a procedure call. The calls are required to

determine if an interrupt has occurred. These particular points have been chosen for two reasons. First, the value of EVRES (the last value computed) is expendable, and thus it does not have to be saved before evaluating a response form. Second, it is not possible to construct a FORM whose evaluation does not ultimately result in a call to ALLOW\INTERRUPT. Therefore, the evaluator will always respond to interrupts. The ability to interrupt a path, perform an arbitrary computation, and then allow the path to continue evaluation is desirable in an interactive implementation of the language. Hence, these checks would probably be included even in a single-path implementation.

The control primitive GOTO must scan the control stack if the path is in the midst of an interrupt response or if it is a supporting path. In either case, a path-dependent flag (IFLG or SPATH) is set. If neither is set (the normal case,) then GOTO may perform the transfer of control without any additional checks. We note that a compiled local GOTO (i.e. within a block) would not even have to check the flags since the environment of the path cannot change.

From the discussions above it should be clear that the inclusion of the multi-path facility in the language causes no significant change in the time and space requirements of a single path of control.

In considering the efficiency of the control primitives themselves, we will restrict ourselves to those primitives which are used most heavily in the examples of chapter 3, namely, PAP and CIA. Many of the others have either trivial implementations or simply involve the modification of tables.

In the implementation of PAP, it is necessary to copy the arguments to the PAPed procedure from the stacks of one path to another. Although this may seem inefficient, we note that the arguments are usually pointers to objects in the heap, and thus they are inexpensive to copy. In any case, the overhead is not significantly greater than if the procedure took all of its arguments BYVAL. If DPAP is used, large stack objects can be passed BYREF without a copy being made. Some copying can be avoided, and thus PAP made more efficient, if the two paths are tied together for the duration of the PAP, i.e. the arguments are pushed directly onto the name and value stacks of the path being PAPed into while the name stack of the original path is used to provide an environment.

Turning to CIA, the primary issue is the amount of work necessary to switch contexts, i.e. how difficult is it for the evaluator to save the state of the current path and then install the state and commence the evaluation of another? Context switching is achieved by pushing the label of a

return statement onto the control stack, saving the three stack pointers and some path dependent flags in the path's ACTRC, loading the new path's stack pointers and flags and then passing control to the return statement specified by the label on the control stack of the new path. Since the 'state' of a path is described by a small set of variables, context switching is relatively inexpensive.

One other issue relating to the CI must be discussed. Since it acts as a single access resource with respect to CIA calls by paths, there exists the possibility that the CI may become a system bottleneck, i.e. processors will waste much of their time in busy waits upon the CI. The situation is similar, in essence, to the use of a global interlock to control access to the traffic controller in a multi-processor system, c.f. 1.2.3. In the latter case, it has been found that a single interlock strategy is superior to one in which many interlocks are used to permit simultaneous access to the controller [Ra68].

Since MPEL1 will usually be implemented in the context of an operating system, busy waits can be replaced with non-busy waits, c.f. 5.2.3. However, the question remains as to how often an evaluator will find the CI busy. Madnick's [Ma68] results show that for a small number of evaluators the probability that the CI is busy is roughly proportional to the number of evaluators and to the fraction

of time each spends executing in the CI environment. For example, if there are three evaluators and each spends (on the average) 5 percent of its time in the CI, then the probability that an evaluator will find the CI busy on any given CIA call is .15. Of course, the fraction of time spent in the CI depends heavily upon the program being run, i.e. on the complexity of the procedures which are CIA called and the number of times they are called in relation to non-CI evaluation.

The use of separate stacks for each path requires justification. If the number of paths is small, then this is not unreasonable. In addition, it is possible to initialize the path with a small stack and allow for expansion as required, c.f. 5.2.1. However, if many control paths are defined, the amount of storage required can become quite large. There are two alternatives. The first is to completely abandon the stack, as in OREGANO [Be71]. The second is to have all paths use a single stack [Bo72]. In the former case, storage for the path's environment is allocated from the heap and managed using a retention strategy, i.e. by garbage collection or reference counts. This technique imposes substantial and unnecessary overheads upon single path evaluation, as the stack discipline is sufficient but must be replaced by the less efficient garbage collection mechanism. Thus, it is unacceptable. The latter solution is quite attractive and

will be discussed further in section 5.4.

Finally, we consider the facility's ability to synthesize multi-path control structures. The examples in chapter 3 demonstrate the range of the facility. In particular, most of the control structure found in the languages discussed in section 1.2.1 are included. Here, we will compare MPEL1 with Fisher's CDL [F170], c.f. 1.2.4.

Fisher claims that as far as he has been able to determine, his primitives "constitute a basis for the mechanisms underlying control structures." If Fisher's primitives can be synthesized or are primitive already in MPEL1, then we can expect that MPEL1 also constitutes a reasonable basis. We shall see. The CDL primitives were described in section 1.2.4 and will not be defined again here.

The primitives seq and cond are subsumed by the EL1 block, i.e.

$$\text{seq}(s1, \dots, sn) = []s1; s2; \dots; sn[]$$

and

$$\text{cond}(p1, e1, \dots, pn, en) = []p1 \Rightarrow e1; \dots pn \Rightarrow en []$$

The primitive par can be defined as a variation on FORK, c.f. 3.3. synch is a variation on TSET, viz.

$$\text{synch}(I, e1, e2) =$$

```

BEGIN
  NOT TSET(I) => e1;
  e2;
  CLEAR(I)
END
```


If several synch operations (with the same first argument) are evaluated simultaneously, then only one will evaluate e_2 , all others will evaluate e_1 . monitor and cont are described in section 3.5. Thus, all of Fisher's primitives may be realized in MPEL1.

Let us examine the MPEL1 definition of cont. To allow a path, say P , to evaluate relatively continuous to all others, it is necessary to interrupt all other active paths and then wait for them to queue themselves before allowing P to continue. A somewhat similar definition of cont is given by Thomas [Th71]. The amount of processing required to implement cont in both MPEL1 and PGL (Thomas' language) raises a question as to whether it should be defined as primitive or obtained by extension. Of course, a clear and precise formal definition of cont (which specifies an efficient realization) would make it an acceptable primitive. We will return to this topic in the next section.

3.2 The Formal Definition

The formal specification of MPEL1 consists of a description of one of n identical evaluators. The evaluator is always processing some MPEL1 control path but not

necessarily the same path at all times, i.e. it may switch its attention from path to path. Because of this context switching, the evaluator must be reentrant with respect to the paths it evaluates. An examination of the procedures which constitute the evaluator reveals that it is written essentially in EL1, i.e. it utilizes only a small number of the control primitives, namely, TSET, CLEAR, EVAL, and GOTO. The question arises as to why the control primitives and framework of MPEL1 are not included in the meta-language; if the primitives are to be used in the synthesis of multi-path control structures why are they not utilized in their own description? Let us consider such a model. In the following discussion, we will use CI', CIA' and FAP' to denote uses of these terms in the meta-language.

The multi-path organization to be described is one in which exactly n interpreter paths are to be evaluated concurrently. For each MPEL1 path there will be an interpreter path (ipath) which evaluates it. Because the correspondence is one-to-one, an ipath may record a path's intra-path control in its own environment. In particular, one ipath is dedicated to the evaluation of the CI path. Only the ipaths are paths with respect to the CI'. The ipaths use the CI' to insure that exactly n of them are evaluated concurrently and to synchronize the activation of the ipath corresponding to the CI (ipath/CI). For example,

if a path, say P, wishes to perform a CIA call, its interpreter (ipath/P) performs a CIA'. If ipath/CI is not currently running, then it is passed control in order to evaluate the CIA call. If ipath/CI is currently running, i.e. if it is either active or on the INACTIVEQ', then ipath/P is queued and an idling path run in its place. When ipath/CI completes its current CIA call, the idling path is stopped and the CIA call for P is executed by passing control to ipath/CI. To PAP a procedure call into the environment of P, an ipath modifies the environment of P and then uses PAP' to force ipath/P to call the procedure application routine.

A number of defects with this model should be immediately obvious. First, to describe the primitives CIA and PAP is necessary to utilize CIA' and PAP', respectively. Thus, there is a direct circularity in the definitions of these primitives. As these are two of the more unusual control primitives, the circularity is objectionable. For example, with two different interpretations of the primitives in mind, one could consult the definition and find both confirmed. In addition, one cannot determine if the primitives have a reasonable implementation from their description in the model. Second, the model requires the use of the CI' in a rather sophisticated way, namely, to insure that exactly n ipaths are evaluated concurrently. In particular, a special scheduling algorithm is required.

However, this implies that CI extensions must be used in the definitions of the primitives. Since such extensions are usually more sophisticated than the primitives themselves, we are in the position of defining reasonably simple concepts in terms of more complex ones.

Let us contrast this model with the one presented in chapter 4. Only four control primitives are used. Of these, two (TSET and CLEAR) can be implemented in a single machine instruction. EVAL is defined trivially as a CALL to EVAL\FORM. GOTO, however, requires some discussion as it is used in a number of ways. First, it is used to 'linearize' control within the evaluator, i.e. to insure that all CALLs to EVAL\FORM occur at the statement level of the block so that no information about the path is retained implicitly in the logic of the evaluator. Second, it is used to return control to statements whose labels have been pushed onto the control stack. Here, control is always returned to a statement in the outermost block of the procedure EVALUATOR. Finally, it is used in its own definition to transfer control to EVALK1 after explicitly flushing the path's stacks and installing the appropriate statement to be evaluated in the BLOCK\BLOCK. Although the definition is circular, note that the general GOTO is defined in terms of explicit modifications to be made to the path's environment and then a local jump to another statement in the same block. Hence, the direct circularity actually lies in the

local GOTO. As local jumps are a fundamental notion and have a straightforward implementation, the direct circularity is acceptable.

In the MPeL1 model, CIA and PAP are both described in terms of the four primitives discussed above and in terms of modifications to the stacks which constitute the path's environment. Thus, there can be no misinterpretation of their semantics or question of the feasibility of their implementation.

To emphasize that we have not been merely raising a strawman, let us consider one of the formal definitions of Fisher's primitives, which were discussed in section 1.2.4.^{*} Here, we are concerned with the second of the three which is a recursive evaluator similar in spirit to the original EL1 definition or a LISP definition of LISP. The control structure of the path is implicit in the environment of the evaluator. Thus, the MPeL1 evaluator outlined above might have taken a similar form. The primitives cont, synch and monitor are defined by direct circularity. For example, to evaluate the form y, where

$$y \leftarrow (\text{CONT } x)$$

the evaluator essentially executes

$$\text{cont (eval(Y.CDR.CAR))}$$

*

The first is an English language description and the third is only valid in a single processor environment.

Thus, any number of different semantics could be associated with cont and all would be equally valid. In addition, we cannot determine if cont has a feasible implementation, or if it is implementable. To illustrate, using direct circularity we could define the primitive $TMHALT(M,T)$ which returns TRUE if and only if Turing machine M halts when given tape T. $TMHALT$ would be an extremely powerful primitive, but unfortunately it cannot be implemented.

In the formal definition of MPEL1, all of the data structures which constitute a path's environment are represented explicitly. The control primitives are defined as operations on these structures. This implies that the model is implementation dependent, i.e. the primitives are described in terms of a preferred implementation. Of course, a precise implementation independent model would be equally as valid. However, as we have seen above, there is often a question as to the feasibility of control primitives. Thus, it is especially important for control that the model be as realistic as possible. To facilitate this, only those control primitives which are intuitively acceptable should be used in the model - the remaining ones must be explicated therein.

The implementation oriented nature of the model offers additional benefits as well. First, it allows the language designer to judge the efficiency of a primitive from its

treatment in the model. This was especially helpful in determining the effects of the primitives upon the evaluation of a single path, as discussed in the previous section. Second, a machine language interpreter for MPeL1 can be coded directly from the model, without mentally 'de-recurring' the evaluator.

This last point requires additional comment. Since the intra-path control is represented explicitly, the EL1 evaluator cannot be represented as a set of procedures which call each other recursively as in [Weg70]. At first glance, it would seem that the bookkeeping required to maintain the necessary information on the control stack would make the model inelegant and unreadable. Here, however, the data definition facility of EL1 proved to be invaluable. In particular, the use of control modes, c.f 4.1.2, allows the necessary items to be gathered into one structure which is then pushed onto the stack. The components of the object may then be referenced by symbolic field names. This allows a sub-evaluator to be written almost as if the field names were the arguments to the sub-evaluator called as a procedure.

The mode STACK was introduced in section 4.1.3 primarily to allow EL1 to be used as the meta-language of the model. It is intended that ordinary LIFO stacks will be used in an actual implementation of MPeL1. Thus, we must

determine whether the stack operations are actually implementable. PUSH presents no problem as it simply pushes an object onto the top of the stack. HEAP is trivial to implement if the address space is divided into segments, some of which are used for heap and the remainder used for stacks. If a table of the segment assignments is maintained, then HEAP reduces to a simple address calculation. In the model, we use integers to index objects on the stack. In an implementation, these will be replaced by actual stack pointers. In this light, FLUSH simply resets the stack pointer and INSTACK determines if the pointer in question lies between the two stack pointers.* The only question remaining is whether or not it is reasonable to index the stack as if it were a ROW. An inspection of the model will show that only the control-stack is so indexed.** In particular, it is referenced in only two ways. First, one of the topmost K objects is referenced, where the modes of the top K objects are known. Second, the stack is searched, starting from the current value of CP. In the former case, since the objects

*

This is still possible even if the stack is allocated in segments, since INSTACK is also given a reference to the stack itself.

**

The name-stack is also indexed, but it is defined as a ROW in the model. The value-stack is indexed only in COPY, where the entire stack is copied, c.f. 5.2.1.

are of fixed size, an appropriate offset can be determined. The latter case is also reasonable since objects pushed onto the control stack are described by a finite set of modes, i.e. they are either SYMBOLs or some control mode. Various encodings can be used to distinguish the objects. For example, the mode and size of the object could also be saved on the stack. More efficient encodings are certainly possible.

We conclude this section with a discussion of the treatment of interrupts in the model. To send an interrupt to an evaluator, a flag associated with the evaluator is set. The evaluator checks this flag at certain points via CALLs to ALLOW\INTERRUPT, c.f. 5.3.1. Of course, in an implementation the interrupts do not actually occur at these nice (or clean) points. Typically, an interrupt can occur after any memory reference.* Since our model uses a high-level programming language, such interruption is below our level of discourse.

When a real interrupt occurs in an implementation of MP-EL1, two different actions may be taken. First, the response to the interrupt may be delayed until the evaluator is willing to accept the interrupt. Alternatively, the interrupt response can be initiated at the time the

*

Note that hardware usually accepts interrupts only at clean points as well, although at a much finer level.

interrupt occurs. There are problems with the latter case (especially if the response form is written in MPEL1.) First, the evaluator may be in the midst of switching contexts (paths.) Thus, there is no environment to evaluate in. Second, even if there is an environment, the stacks may not be in reasonable enough shape to allow the evaluation of an arbitrary form. In particular, if the response form invokes a garbage collection, then heap objects referenced only by the machine registers could be lost. Some of these problems can be alleviated by specifying that portions of the system code are not interruptible, i.e. that the interrupt must wait till the next clean point. In general, we would expect that hard interrupts of this sort would perform the minimal necessary operations and then generate a lower priority processor level interrupt to continue the processing at the next convenient point, namely, at the next call to ALLOW\INTERRUPT.

4. CONCLUSIONS AND SUGGESTIONS FOR FUTURE RESEARCH

This dissertation has investigated the problem of introducing multi-path control structures into programming languages. The approach taken has been to define a set of control primitives and a language framework from which various multi-path organizations can be realized by extension. The primitives are cast in a multiple processor environment to avoid any bias towards the single processor case.

Our basic assumption has been that there are n physical processors and k paths of control. We have shown that various multi-path organizations and operations can be described simply as specifications of the way in which the processors are to be assigned to paths. The control interpreter path allows the user control over this assignment, and thus gives him the ability to synthesize multi-path control structures. Two properties of the CI facilitate this. First, all control transfers between paths must be via the CI. Thus, it can keep track of the processor-path assignments. Second, only one path may pass control to the CI at a time. When a path transfers control to the CI, using CIA, it specifies a procedure to be evaluated in the CI environment. The evaluation of this procedure is indivisible with respect to other CIA calls of the same procedure (or others.) Thus, such procedures are

given an environment and a mode of operation in which they can examine and modify the processor-path assignments. In addition, since only the existence of the CI is assumed by the primitives, it is possible to redefine or extend the control interpreter in MPEL1.

In addition to providing a precise specification of the semantics of the control primitives, the formal definition of MPEL1 has yielded a number of other benefits. First, we were able to determine that the control primitives are pragmatically valid since the primitives of the model can be realized on contemporary hardware. Second, because the model is implementation-oriented we were able to use it to assess the effect of the multi-path facility upon single path evaluation. Finally, it demonstrates the values of including the intra-path control as an explicit structure in the meta-language.

Although we believe that MPEL1 constitutes a reasonable basis for a multi-path facility, a number of additions and generalizations can be made. Some of these are outlined below.

Recently, Bobrow and Wegbreit (B-W) [Bo72] have developed a technique which allows multiple paths of control to be implemented on a single stack. If only a single control path is used, then it runs as efficiently as it would if it was assigned its own separate stack. Although

the use of separate stacks, as in MPEL1, is more reasonable than a heap-garbage collection scheme, a single stack is even more attractive. In particular, it avoids a number of the overheads associated with stack expansion, as discussed in section 5.2.1. Thus, it would be desirable to incorporate the B-W technique into the multi-path facility.

It is interesting to note that the description of the B-W technique uses a model similar to the formal model of MPEL1, in that the paths' call structures are represented explicitly. Although B-W give an English language description, a formal specification of a programming language which utilizes their technique would thus take the form of the MPEL1 definition.

MPEL1 assumes that the n processors can all readily access a common address space. This usually implies that the processors are in close proximity to one another. The construction of the ARPA network [Ro70] allows one to consider distributed computations in which paths of control are evaluated by processors on different nodes of the net. This raises a number of questions. For example, how can data structures be shared effectively across many nodes? How does one represent pointers to objects on other nodes? In particular, how can garbage collection be effected in the distributed environment? For our purposes, we are primarily concerned with the modifications which must be made to the

CI and the control primitives. Should there be a CI for each node or one master CI that resides on a single node? Although the latter case is a straightforward extension of the current CI, transfers of control between paths on a node which does not contain the CI become rather slow. The former solution avoids this problem, but the absolute indivisibility of CI execution (with respect to paths on all nodes) is lost. The primitives that access environments of other paths (e.g. PFETCH, DEPEND) may have to utilize the CI in order to allow the processor to be reassigned while the information is being transmitted from a path on another node.

Except for the addition of a few intra-path control primitives, single path MPEL1 is essentially EL1. We have left this constant in order to focus on inter-path issues. However, we believe that work remains to be done in the area of intra-path control structures. Here, we refer to the possibility of providing extension mechanisms which allow for the compilation of control and environments. Most languages do not allow for variation in the environmental structures to be used in the evaluation of a program. Why is this so? We believe that there are two reasons. First, the most fundamental semantics of the language, e.g. the scope rule, recursion, etc. restrict the class of structures which may be used. Second, a compiler must know what the environmental structures are in order to generate

code. Allowing variation in these structures makes compilation difficult, if not impossible.

However, control over environmental structures can be useful. For example, if a set of EL1 procedures reference the same top-level variables and do not utilize recursion, then their evaluation would best be effected using a FORTRAN-like environmental structure in which storage for locals and globals is only allocated once.

The problem can be approached in two ways. First, the compiler can be made 'smart' in order to deduce the appropriate environmental structures. This requires techniques from the field of program automation [Ch71]. Alternatively, a set of environmental primitives can be included in the language. Using these, the programmer may synthesize the environmental structures desired. The compiler can then be appropriately parameterized to utilize information provided by these primitives.

Appendix 1

INTRODUCTION TO EL1

The purpose of this appendix is to provide a brief introduction to EL1 for those readers who are unfamiliar with the language. It does not attempt to give a complete description of EL1. We direct the reader who desires a more precise description to either [Weg70], [Weg72], or Chapter 4.

At a superficial level, EL1 appears to be a conventional programming language in the spirit of ALGOL 60. It includes variables, arrays, assignments, procedure calls, prefix and infix operators, block structure, labelled statements and gotos. For example, all of the following are legal in EL1.

```
Q ← SQRT(A)/(R[3]-Y)
L: BEGIN
    X ← FUM(A,B) - Z;
    FI(X,R[I])
END;
B ← C OR D;
```

These examples, however, are a bit misleading in that they are special cases of more general syntactic forms. To illustrate, assignment is treated as a right-associative binary operator whose value is its left hand operand.

Blocks have values — the value of the last statement evaluated.

The most basic unit in EL1 is the form. Examples of forms are:

- (1) constants such as 3 and FALSE,
- (2) variables such as Y and NAME\INDEX,
- (3) expressions composed of infix and prefix operators such as A+B, NOT D OR E[2],
- (4) selections of compound objects such as PAVECT[I] and P.CIA\RESULT,
- (5) procedure calls like SUSPEND(P) and ENTERL(PAVECT[PROCNUM],INACTIVEQ).

More complex forms may be constructed by combining forms according to the syntax rules of the language. In general, a form is a syntactically complete unit which may be evaluated to yield a value. An EL1 program is a form which is not contained as part of a larger form.

A block is a form which is composed of a sequence of statements. Each statement of a block is either a form or one of two types of conditionals, viz.

f -> g;

f => g;

In the former case, the interpretation is that if f is TRUE, then g is evaluated and execution continues with the next statement in the block. In the latter case, if f is TRUE

then g is evaluated and the block is exited with the value of g taken as the value of the block. For example, if V is a vector of N integers, the following block computes the sum of the positive elements.

```

BEGIN
  I ← 1;
L: V[I] > 0 → S ← S + V[I];
  I = N ⇒ S;
  I ← I+1;
  GOTO L
END;

```

Variables are either top-level variables, formal parameters to a ROUTINE, or declared variables local to a block. In all cases, a variable must be declared to be of some specific data type and it may only contain values of that type throughout its lifetime. If a variable is used in a block but is not defined therein, then it is said to be free. EL1 uses a dynamic scope rule for the identification of free variables, i.e. the value of a free variable is the value of the most recently created variable with that name in the dynamic call structure of the program.

EL1 contains a number of built-in data types, called modes, and a number of mode constructing operators which allow for the creation of new data types, as required. The built-in modes include BOOL (Boolean), CHAR (character), INT (fixed-point integer), SYMBOL (non-numeric atom as in LISP [We67]), MODE (the data type of data types), REF (pointers to objects of any mode), FORM (similar to LISP

S-expressions), and ROUTINE (user-defined or built-in procedure). There are four mode constructing operators. Each one yields a MODE value which may be assigned to a variable of mode MODE. The operators are best explained by example.

The operator ROW constructs modes for arrays of homogeneous objects. For example, the statements

```
I3 ← ROW(3,INT);
```

```
BOOLR ← ROW(BOOL);
```

assign to I3 and BOOLR the modes 'array of three integers' and 'array of any number of Booleans', respectively. Variables may be declared using these modes as follows.

```
DECL V:I3;
```

```
DECL B:BOOLR BYREF CONST(BOOLR SIZE 4);
```

V is a ROW of three integers. The length of the ROW is fixed by the definition of I3. B is a ROW of four BOOLs. Here, the length of the ROW is resolved at the time B is created by specifying an initial value, and hence a length, for the variable. The operator CONST will be discussed in more detail below.

The operator STRUCT constructs modes for compound objects whose components do not necessarily have the same mode. For example, the statement

```
ENV\BLOCK←STRUCT(OLD\NP:INT,OLD\VP:INT,RETURN:SYMBOL);
```

assigns to ENV\BLOCK the data type for a structured object.

An object of mode ENV\BLOCK consists of three components:

- (1) an INT named OLD\NP,
- (2) an INT named OLD\VP,
- (3) a SYMBOL named RETURN.

The operator PTR constructs modes for objects which may point to other objects. The arguments to PTR specify the modes of the objects to which an object of the new mode may point. E.g.,

```
I\OR\B\PTR <- PTR(INT, BOOL);
```

An object of mode I\OR\B\PTR may point to either an integer or a Boolean. The operator ALLOC is used to create objects of mode class PTR. ALLOC is discussed later in this section.

The operator ONEOF constructs modes which represent one of a set of modes. For example, the statement

```
I\OR\B <- ONEOF(INT,BOOL);
```

assigns to I\OR\B the mode 'one of the modes INT or BOOL'. It is not possible to construct an object of mode I\OR\B, i.e. at the time a variable of mode I\OR\B is created, it must be type resolved to either an INT or a BOOL. E.g.,

```
DECL X:I\OR\B BYREF CONST(INT);
DECL Y:I\OR\B BYREF CONST(BOOL);
```

X is of mode integer and Y is a Boolean. ONEOF may be used to specify that an argument to a procedure may be an object of one of a set of modes, c.f. 2.3.1. The built-in mode ANY may be described as ONEOF('any mode').

It should be noted that the arguments to the mode operators may themselves be calls upon the operators. For example, if N and M are integers, then a mode for N by M integer matrices can be constructed as follows.

```
ROW(N,(ROW(M,INT)));
```

The i th element (for i between 1 and N) of an object of this mode is an M element ROW of integers.

The components of ROWs and STRUCTs may be selected. The components of ROWs may only be selected by integer indices. If V is an I3, then

```
V[1+1]
```

selects the second integer of the ROW. The components of STRUCTs, however, may be selected either by integer indices or by their symbolic field names. If E is an ENV\BLOCK, then all of the following select the RETURN component.

```
E.RETURN  
E["RETURN"]  
E[3]
```

Objects of mode class PTR may also be selected in the sense that the pointer is followed until a ROW or STRUCT is found and then the selection is performed on the compound object. For example, if P is a PTR(ENV\BLOCK) then

```
P.RETURN
```

selects the return component of the ENV\BLOCK.

A pointer may be followed explicitly by using the operator VAL. The value of VAL is the object referenced by

the pointer. If Q is an PTR(INT), then

```
VAL(Q) ← 3;
```

assigns 3 to the integer referenced by Q.

Variables are distinct from objects in EL1. Each variable names some object. However, an object may be named by more than one variable. In addition, several variables may name different components of a single object. In EL1, an object may lie either on a block structured stack or in a free storage region called the heap. In the former case, the lifetime of the object is the same as that of the block in which it was created. In the latter case, the object exists as long as it can be referenced.

An object is created either implicitly as the result of a declaration or explicitly via calls to the object generators CONST and ALLOC. In either case, the mode of the object is fixed at the time of creation and the object retains the mode throughout its existence. For example, the three statements

```
DECL P:ENV\BLOCK;
DECL Q:ENV\BLOCK BYREF
      CONST(ENV\BLOCK OF 1,2, "DELPTH");
DECL T:I\OR\B\PTR;
```

generate three stack objects. P is initialized as an ENV\BLOCK with default values for its components. Q is initialized as an ENV\BLOCK with the components 1, 2 and "DELPTH", respectively. T is initialized as a null I\OR\B\PTR.

If a stack object is returned as the value of the block or procedure application in which it was created, then it becomes a pure-value in the sense that assignments to it are harmless, but useless. For example, consider the following

```
BEGIN
  DECL X:INT;
  [ ) DECL Y:INT; P=> Y ; X ( ] <- 4;
  X+2
END;
```

If P is FALSE, then the value of the inner block is X, which is then assigned the value 4. If, however, P is TRUE, then the value of the block is Y, which is converted to a pure-value. Assigning 4 to the pure-value has no effect upon the future evaluation of the program.

The generator ALLOC is similar to CONST, except that the object generated is allocated in the heap. ALLOC returns a pointer to the object generated. For example,

```
DECL P:I\OR\E\PTR;
```

```
·
·
·
```

```
P <- ALLOC(INT LIKE 1);
```

The I\OR\E\PTR P is constructed on the stack and is initialized to NIL. An integer is allocated in the heap and P is assigned a pointer to the integer. The value of the integer may be accessed using the operator VAL, which has been described earlier.

In EL1, ROUTINES subsume the notions of procedures and

operators. A variable of mode ROUTINE may be assigned a procedure value, viz.

```

FIB <- EXPR(X:INT; INT)
  BEGIN
    X = 0 => 1;
    X = 1 => 1;
    FIB(X-1) + FIB(X-2)
  END;

```

The ROUTINE FIB computes the N th element in the Fibonacci series. FIB takes a single formal parameter named X. The mode of X is INT. The bind class of X is defaulted to be BYREF (by reference,) i.e. an assignment to X would change the value of the argument. The result type of the procedure is INT, i.e. the mode of the object returned by the procedure is INT.

A call to a ROUTINE may be written as a function name followed by an argument list:

```
FIB(ENV\BLOCK.OLD\NP)
```

A ROUTINE valued variable can also be declared to be a NOFIX, PREFIX or INFIX operator as it takes zero, one or two arguments, respectively. In the first two cases, the routines may be called without enclosing their arguments in parentheses. In the last case, the arguments appear to

*
Other bind classes in EL1 are BYVAL, UNEVAL, and LISTED. If BYVAL is used, then the formal is bound to a copy of the actual parameter. UNEVAL and LISTED may only be used if the mode of the formal is FORM. With UNEVAL, the formal parameter is bound to the unevaluated list structure for the actual. With LISTED, the formal is bound the remaining argument list.

either side of the operator. For example,

```

FOUR <- EXPR(; INT) 4;
FUM <- EXPR(X:INT, Y:INT; INT) (X+Y+2);
PREFIX(FIB);
NOFIX(FOUR);
INFIX(FUM);
.
.
.
FIB FOUR ; NT Same as FIB(FOUR());
1 FUM 2 ; NT Same as FUM(1,2);

```

Top level variables may be assigned values without explicit declaration. The first assignment to the variable determines its mode. For example

```
X <- 1 + 1;
```

declares X to be of mode INT and binds it to the integer 2. Subsequent to the assignment, X may only be assigned integer values. The mode of X can only be changed by calling the built-in routine FLUSH. E.g. FLUSH(X) disassociates X from its mode and value. X may then be assigned a new value (and mode), e.g.

```
X <- TRUE;
```


Appendix 2

SYNTAX OF EL1

The concrete syntax of EL1 is specified by a BNF grammar. Non-terminals of the grammar are sequences of characters enclosed in the brackets $\langle \rangle$. All other symbols, except for $::=$ and $'|'$, are terminals of the grammar. The rules of the grammar are of the form

$$\langle \text{NT} \rangle ::= \underline{A}$$

where \underline{A} denotes a string of terminals and non-terminals. For compactness, the rules

$$\langle \text{NT} \rangle ::= \underline{A_1}$$
$$\vdots$$
$$\langle \text{NT} \rangle ::= \underline{A_n}$$

are abbreviated as follows.

$$\langle \text{NT} \rangle ::= \underline{A_1} \mid \underline{A_2} \mid \dots \mid \underline{A_n}$$

The abstract syntax representation of an EL1 program is a list structure. The correspondence between the concrete and abstract representations of EL1 is specified by augments to the BNF grammar. In each of the rules below, the augment is separated from the right hand side of the production by the symbol $'@'$. An augment specifies the actions to be taken when the corresponding reduction is made during the

parse. There are four different formats for augments. Their interpretations are best explained by example.

The augmented rule

$\langle \text{form9} \rangle ::= \langle \text{selection} \rangle @ \langle \text{selection} \rangle$

specifies that in reducing a $\langle \text{selection} \rangle$ to a $\langle \text{form9} \rangle$, the list structure associated with the $\langle \text{selection} \rangle$ is to be associated with the $\langle \text{form9} \rangle$ directly.

The augmented rule

$\langle \text{selection} \rangle ::= \langle \text{form3} \rangle . \langle \text{id} \rangle @ (\text{SELQ! } \langle \text{form3} \rangle \langle \text{id} \rangle)$

specifies that the list structure to be associated with the $\langle \text{selection} \rangle$ is obtained by constructing a three element list. The first element is the SYMBOL SELQ!. The second and third elements are the list structures associated with the $\langle \text{form3} \rangle$ and the $\langle \text{id} \rangle$, respectively.

The augmented rule

$\langle \text{str-form} \rangle ::= \text{STRUCT}(\langle \text{structlist} \rangle) @ \text{STRUCT} \& \langle \text{structlist} \rangle$

specifies that the list structure to be associated with the $\langle \text{str-form} \rangle$ is obtained by CONSing the SYMBOL STRUCT onto the head of the list associated with the $\langle \text{structlist} \rangle$. & is a right-associative infix operator equivalent to CONS.

The augmented rule

$\langle \text{fmstr} \rangle ::= \langle \text{fmstr} \rangle , \langle \text{form} \rangle @ \langle \text{fmstr} \rangle \leftarrow \& \langle \text{form} \rangle$

specifies that the list structure to be associated with the $\langle \text{fmstr} \rangle$ is obtained by placing the $\langle \text{form} \rangle$ at the end of the

list specified by $\langle \text{fmstr} \rangle$. For example, if the $\langle \text{form} \rangle$ is 4 and the $\langle \text{fmstr} \rangle$ is (1 2 3), then the resulting $\langle \text{fmstr} \rangle$ is the list (1 2 3 4).

In an augment, the expression $\langle \text{NT} \rangle[i]$ specifies the i th element of the list associated with $\langle \text{NT} \rangle$ and the expression $\langle \text{NT}-i \rangle$ specifies the i th occurrence of the non-terminal $\langle \text{NT} \rangle$ in the corresponding production.

The non-terminals $\langle \text{id} \rangle$, $\langle \text{constant} \rangle$, $\langle \text{prefixop} \rangle$, and $\langle \text{infixop} \rangle$ denote the (not necessarily disjoint) sets of identifiers, constants, prefix operators and infix operators, respectively.

The grammar follows.

$\langle \text{program} \rangle$	$::=$	$\langle \text{form} \rangle$	@ $\langle \text{form} \rangle$
$\langle \text{form} \rangle$	$::=$	$\langle \text{iteration} \rangle$ $\langle \text{fn-call} \rangle$ $\langle \text{exprnt} \rangle$	@ $\langle \text{iteration} \rangle$ @ $\langle \text{fn-call} \rangle$ @ $\langle \text{exprnt} \rangle$
$\langle \text{form9} \rangle$	$::=$	$\langle \text{constant} \rangle$ $\langle \text{id} \rangle$ BEGIN $\langle \text{blockbody} \rangle$ END $\langle \text{mform} \rangle$ $\langle \text{selection} \rangle$ $\langle \text{generation} \rangle$ ($\langle \text{form} \rangle$)	@ $\langle \text{constant} \rangle$ @ $\langle \text{id} \rangle$ @ BLOCK! & $\langle \text{blockbody} \rangle$ @ $\langle \text{mform} \rangle$ @ $\langle \text{selection} \rangle$ @ $\langle \text{generation} \rangle$ @ $\langle \text{form} \rangle$
$\langle \text{blockbody} \rangle$	$::=$	$\langle \text{declstr} \rangle$; $\langle \text{stat} \rangle$ $\langle \text{stat} \rangle$ $\langle \text{blockbody} \rangle$; $\langle \text{stat} \rangle$	@ $\langle \text{declstr} \rangle$ <-& $\langle \text{stat} \rangle$ @ ($\langle \text{stat} \rangle$) @ $\langle \text{blockbody} \rangle$ <-& $\langle \text{stat} \rangle$
$\langle \text{declstr} \rangle$	$::=$	$\langle \text{declnt} \rangle$ $\langle \text{declstr} \rangle$; $\langle \text{declnt} \rangle$	@ ($\langle \text{declnt} \rangle$) @ $\langle \text{declstr} \rangle$ <-& $\langle \text{declnt} \rangle$

<declnt>	::=	DECL <idstr> : <form> DECL <idstr> : <form> <initd>	@ (DECL! <idstr> <form>) @ DECL! & <idstr> & <form> & <initd>
<idstr>	::=	<id> <idstr> , <id>	@ (<id>) @ <idstr> <-& <id>
<initd>	::=	BYVAL <form> BYREF <form>	@ { BYVAL <form> } @ { BYREF <form> }
<stat>	::=	<form> <form> -> <form> <form> => <form> <id> : <stat>	@ <form> @ { IF! <form-1> <form-2> } @ { CLAUSE! <form-1> <form-2> } @ (LABST! <id> <stat>)
<iteration>	::=	FOR <id> <- <fmit> DO <form>	@ (FOR! <id> <fmit>[1] <fmit>[2] <fmit>[3] <fmit>[4] <form>)
<fmit>	::=	<fmit1> <fmit1> <test>	@ <fmit1> <-& NIL @ <fmit1> <-& <test>
<fmit1>	::=	<form> , ... , <form> <form> , <form> , ... , <form>	@ { <form-1> NIL <form-2> } @ { <form-1> <form-2> <form-3> }
<test>	::=	WHILE <form> TILL <form>	@ { WHILE . <form> } @ { TILL . <form> }
<mform>	::=	<mform2> <id> : : <mform2>	@ (<mform2>[1] NIL <mform2>[2]... <mform2>[n]) @ (<mform2>[1] <id> <mform2>[2]... <mform2>[n])
<mform2>	::=	ROW (<form>) ROW (<form> , <form>) PTR (<fmstr>) ONEOF (<fmstr>) <str-form>	@ (ROW NIL <form>) @ (ROW <form-1> <form-2>) @ PTR & <fmstr> @ ONEOF & <fmstr> @ <str-form>
<str-form>	::=	STRUCT (<structlist>)	@ STRUCT & <structlist>
<structlist>	::=	<id> : <form> <structlist> , <id> : <form>	@ ((<id> . <form>)) @ <structlist> <-& (<id> . <form>)
<bind-class>	::=	BYREF BYVAL LISTED UNEVAL	@ BYREF @ BYVAL @ LISTED @ UNEVAL

<selection>	::=	<form3> . <id> <form3> [<form>]	@ (SELQ! <form3> <id>) @ (SEL! <form3> <form>)
<init>	::=	LIKE <form> SIZE <fmstr> OF <fmstr>	@ (LIKE <form>) @ SIZE & <fmstr> @ OF & <fmstr>
<fmstr>	::=	<form> <fmstr> , <form>	@ (<form>) @ <fmstr> <-& <form>
<fn-call>	::=	<form2> <infixop> <form> <form2>	@ (<infixop> <form2> <form>) @ <form2>
<form2>	::=	<prefixop> <form2> <form3>	@ (<prefixop> <form2>) @ <form3>
<form3>	::=	<form3> () <form3> (<fmstr>) <form9>	@ (<form3>) @ <form3> & <fmstr> @ <form9>
<exprnt>	::=	EXPR (<expr1> ; <form>) <form9> EXPR (; <form>) <form9>	@ (EXPR! <expr1> <form> <form9>) @ (EXPR! NIL, <form> <form9>)
<expr1>	::=	<id> : <form> <id> : <form> <bind-class> <expr1> , <id> : <form> <expr1> , <id> : <form> <bind-class>	@ ((<id> <form> BYREF)) @ ((<id> <form> <bind-class>)) @ <expr1> <-& (<id> <form> BYREF) @ <expr1> <-& (<id> <form> <bind-class>))
<generation>	::=	<regionspec> (<form> <init>) <regionspec> (<form>)	@ <regionspec> & <form> & <init> @ (<regionspec> <form>)
<regionspec>	::=	ALLOC CONST	@ ALLOC @ CONST

Appendix 3

CI PROCEDURES AND INTERRUPT RESPONSE FORMS

Modes

```
ARQPTR <- STRUCT(FIRST:ARPTR, LAST:ARPTR);
LIST <- PTR(DTPR);
LISTROW <- ROW(NPROC, LIST);
PROW <-
    ROW(NPROC, STRUCT(CURPATH:ARPTR, IDLEPATH:ARPTR));
NT NPROC is defined by INSTALL\GLOBAL\ENV
    to be equal to the number of processors;
```

Procedures

```
NT INIT\CI initializes the CI. Its arguments
    specify the idle paths for the processors and
    the form to be evaluated;

INIT\CI <-
    EXPR(IDLEVECT:ROW(NPROC, ARPTR), PROC:FORM; NONE)
    BEGIN
        DECL IASTRUN:ARPTR;
        DECL INACTIVEQ:ARQPTR;
        DECL NPROC:INT BYREF NPROC;
        DECL PROCNUM:INT BYVAL 1;
        DECL USER\SCHEDULER:ROUTINE
            BYVAL INITIAL\SCHEDULER;
        DECL PAVECT:PROW;
        DECL RUNSET\FLAG:BOOL;
        DECL PIVECT:LISTROW;

        NT Initialize the PAVECT;

        FOR I <- 1, ..., NPROC DO
            PAVECT[I].IDLEPATH <- IDLEVECT[I];

        NT Create a path in which to evaluate PROC;

        IASTRUN <- GET\PATH(1);
        PAPQ(EVAL(PROC), IASTRUN);
```

NT Commence scheduling;

C\I()

END;

C\I <- FXPR(; NONE)

BEGIN

```
DECL LASTRUN:ARPTR BYREF LASTRUN;
DECL INACTIVEQ:ARQPTR BYREF INACTIVEQ;
DECL NPROC:INT BYREF NPROC;
DECL NFPROC:INT BYREF NFPROC;
DECL PROCNUM:INT BYREF PROCNUM;
DECL USER\SCHEDULER:ROUTINE BYREF USER\SCHEDULER;
DECL PAVECT:PROW BYREF PAVECT;
DECL RUNSET\FLAG:BOOL BYREF RUNSET\FLAG;
DECL PIVECT:LISTROW BYREF PIVECT;
```

NT When C\I is initially called, LASTRUN specifies the path to which control is to be transferred and PROCNUM specifies the current processor;

CONTINUE\PATH:

```
PAVECT[PROCNUM].CURPATH <- LASTRUN;
LASTRUN.PRO <- PROCNUM;
```

NT Transfer control to the path;

```
LASTRUN <- CONTIPATH(LASTRUN);
```

NT CONTIPATH returns the ARPTR of the path performing the CIA call;

```
PROCNUM <- LASTRUN.PRO;
RUNSET\FLAG <- FALSE;
```

NT Apply the CIA-called procedure;

BEGIN

```
MD(VAL(LASTRUN.CIA\FN))=ATOM =>
    EVAL(LASTRUN.CIA\FN)(LASTRUN.CIA\ARG);
LASTRUN.CIA\FN(LASTRUN.CIA\ARG)
```

END;

NT If LASTRUN is NIL, then a new path must be scheduled;

```
LASTRUN=NIL -> GOTO NEWPATH;
```

NT If RUNSET\FLAG is FALSE, then simply pass control to LASTRUN;

NOT RUNSET\FLAG -> GOTO CONTINUE\PATH;

NT Otherwise, interrupt an idling processor so
that it may be assigned to a path;

SIGNAL\IDLE\PROCESSOR();

GOTO CONTINUE\PATH;

NT Call the user's scheduler to obtain a path
to be assigned to the processor;

NEUPATH:

BEGIN

DECL B:BOOL;

B <- PAVECT[PROCNUM].IDLEPATH =
PAVECT[PROCNUM].CURPATH;

NT B is TRUE if and only if the current
processor has been idle;

LASTRUN <- USER\SCHEDULER();

NT LASTRUN is NIL if there exist no paths
to be run, otherwise it specifies the path to
be assigned to the processor;

LASTRUN # NIL =>

BEGIN

B -> NFPROC <- NFPROC-1;

NT One less idle processor;

SIGNAL\IDLE\PROCESSOR()

NT Signal another idle processor;

END;

NT Since there exists no path to run and
the processor was idle (B=TRUE), let it
continue to idle;

B => LASTRUN <- PAVECT[PROCNUM].IDLEPATH;

NT Otherwise, the processor was not idle
before the CIA call. Since there
are no paths to run, let it idle;

LASTRUN <- PAVECT[PROCNUM].IDLEPATH;

NFPROC <- NFPROC+1

END;

GOTO CONTINUE\PATH

END;


```

SIGNAL\IDLE\PROCESSOR <- EXPR(;NONE)
BEGIN
  DECL FPROC:INT;

  NT If there exists an idle processor,
    then it is interrupted. Otherwise,
    no action is taken;

  NFPROC=0 => NOTHING;
  FOR I<-1, ..., NPROC TILL FPROC GT 0 DO
    BEGIN
      I=PROCNUM => NOTHING;
      NT Don't consider the current processor;
      PAVECT[I].CURPATH = PAVECT[I].IDLEPATH =>
        FPROC<-I
    END;

    NT Put a form on PIVECT[FPROC] which will cause the
      processor FPROC to call the USER\SCHEDULER when
      FPROC passes control to the CI due to the
      "PRO\PRO" interrupt sent by STOP\PATH;

    PIVECT[FPROC] <-
      CONS(QUOTE(LASTRUN<-NIL),PIVECT[FPROC]);

    STOP\PATH(PAVECT[FPROC].IDLEPATH)
  END;

INITIAL\SCHEDULER <- EXPR(; ARPTR)
BEGIN
  DECL Y:ARPTR;
  Y <-INACTIVEQ.FIRST;
  L: Y=NIL => NIL;
  NOT Y.DORMANT =>
    BEGIN
      REMOVE(Y,INACTIVEQ);
      Y.TICKS\LEFT <- NUMTICKS;
      NT Set the time allocation for the path;
      Y
    END;
  Y <- Y.NEXT;
  GOTO L
END;

ENTERL <- EXPR(P:ARPTR, Q:ARQPTR; NCNE)
BEGIN
  P.NEXT <- NIL;
  Q.LAST=NIL => Q.FIRST<-Q.LAST<-P;
  Q.LAST.NEXT<-P;
  Q.LAST<-P
END;

```

```

REMOVE <- EXPR(X:ARPTR, Y:ARQPTR; NONE)
  BEGIN
    DECL Z:ARPTR BYVAL Y.FIRST;
    X=Z =>
      BEGIN
        (Y.FIRST <- Y.FIRST.NEXT)=NIL => Y.LAST<-NIL;
      END;
    L: Z.NEXT=X =>
      BEGIN
        Y.LAST=X -> Y.LAST<-Z;
        Z.NEXT <- X.NEXT
      END;
    Z <- Z.NEXT;
    GOTO L
  END;

```

Response Forms

NT PRO\PRO\FORM is the "PRO\PRO" interrupt response form. It generates the path level interrupt "CI\TO\PATH" which then CIA calls a procedure which evaluates all forms on the processor's PIVECT list;

```
PRO\PRO\FORM <- QUOTE(INTERRUPT("CI\TO\PATH"));
```

NT CI\PATH\FORM is the response form associated with the "CI\TO\PATH" interrupt;

```
CI\PATH\FORM <- QUOTE(CIA("PROINT"));
```

```

PROINT <- EXPR(L:LIST; NONE)
  BEGIN
    DECL L:LIST BYVAL PIVECT[PROCNUM];
    PROEVAL(L);
    PIVECT[PROCNUM] <- NIL
  END;

```

```

PROEVAL <- EXPR(L:LIST; NONE)
  BEGIN
    L=NIL => NOTHING;
    EVAL(L.CAR);
    PROEVAL(L.CDR)
  END;

```

END;

NT TIMER\FORM is the "TIMER" interrupt response form. It generates the path level interrupt "TIME\OUT" if the path's time allocation has been exhausted;

```
TIMER\FORM <-
  QUOTE(BEGIN
    MYPATH=PCIAR => NOTHING;
    MYPATH.TICKS\LEFT <-MYPATH.TICKS\LEFT-1;
    MYPATH.TICKS\LEFT=0 =>
      INTERRUPT("TIME\OUT")
  END);
```

NT TIME\OUT\FORM is the response form associated with the "CI\TO\PATH" interrupt;

```
TIME\OUT\FORM <- QUOTE(CIA("NOTIME"));
```

```
NOTIME <- EXPR(; NONE)
  BEGIN
    LASTRUN = PAVECT[PROCNUM].IDLEPATH =>
      LASTRUN <- NIL;
    NT Put the path of the INACTIVEQ and set
      LASTRUN to NIL to force scheduling;
    ENTERL(LASTRUN, INACTIVEQ);
    LASTRUN <- NIL
  END;
```

REFERENCES

- [ACM70] Record of the Project MAC Conference on Concurrent Systems and Parallel Computation, ACM, New York, 1970.
- [An65] Anderson, J.P. "Program Structures for Parallel Processing," Comm. ACM Vol. 8, No. 12 (December 1965), pp. 786-788.
- [BBN70] Bolt, Beranek, and Newman, Inc. "TENEX Technical Manual," January 1970.
- [Be70] Beech, D. "A Structural View of PL/I," Computing Surveys Vol. 2, No. 1 (March 1970), pp. 33-64.
- [Ber71] Berry, D.M. "Introduction To OREGANO," in [Wegn71], pp. 171-189.
- [Bo72] Bobrow, D. and Wegbreit, B. "A Model for Stack Implementation of Multiple Environments," Bolt, Beranek and Newman, Inc. Report No. 2334, March 1972.
- [Bu68] Burge, W.H. "McG - A Functional Programming System," Report RC-2111, IBM T.J. Watson Research Center, Yorktown Heights, New York (1968).
- [Ch68] Cheatham, T.E. et al. "On the Basis for ELF - An Extensible Language Facility," Proc. FJCC Vol. 32 (1968), pp. 937-947.
- [Ch72] Cheatham, T.E. and Wegbreit, B. "A Laboratory for the Study of Automating Programming," Proc. SJCC (1972).
- [Chris69] Christensen, C. and Shaw, C.J. (editors) "Proc. of the Extensible Languages Symposium," in SIGPLAN Notices, Vol. 4, No. 8, August 1969.
- [Co63] Conway, M.E. "A Multi-Processor System Design," Proc. FJCC Vol. 24 (1963), pp. 139-146.
- [Co63a] Conway, M.E. "Design of a Separable Transition-Diagram Compiler," Comm. ACM Vol. 6, No. 7 (July 1963), pp. 396-408.
- [Cor65] Corbato, F.J and Vyssotsky, V.A. "Introduction and Overview of the Multics System," Proc. FJCC Vol. 27 (1965).

- [Da66] Dahl, O. and Nygaard, K. "SIMULA - an ALGOL Based Simulation Language," Comm. ACM Vol. 9, No. 9 (September 1966), pp. 671-678.
- [Da70] Dahl, O. et al. "SIMULA 67 Common Base Language," Norwegian Computing Center No. S-22 (October 1970).
- [DeBak67] DeBakker, J.W. Formal Definition of Programming Languages, Mathematisch Centrum, Amsterdam 1967.
- [Di66a] Dijkstra, E.W. "Cooperating Sequential Processes," in Programming Languages, edited by F. Genuys, Academic Press, New York (1968).
- [Di68b] Dijkstra, E.W. "The Structure of THE Multi-Programming System," Comm. ACM Vol. 11, No. 6 (May 1968), pp. 341-346.
- [Fi70] Fisher, D.A. Control Structures for Programming Languages, Doctoral Dissertation, Carnegie-Mellon University, June 1970.
- [Fl67] Floyd, R.W. "Nondeterministic Algorithms," JACM, Vol. 14 (October 1967), pp. 636-644.
- [Ger69] Gerhart, S. A Survey of Extensible Languages, Preliminary draft, The RAND Corporation, Santa Monica, California, August 1969.
- [Ger70] Gerhart, S. Formal Definition of APL, Unpublished paper, Computer Science Department, Carnegie-Mellon University, March 1970.
- [Gar66] Garwick, J.V. "The Definition of Programming Languages by their Compilers," in [Ste66] pp. 139-141.
- [Go65] Golomb, S.W. and Baumbert, L.D. "Backtrack Programming," JACM, Vol. 12 (October 1965), pp. 636-644.
- [Go66] Gosden, J.A. "Explicit Parallel Processing Description and Control in Programs for Multi- and Uni- Processor Computers," Proc. FJCC, Vol. 29 (1966), pp. 651-660.
- [IBM68] "IBM System/360 Principles of Operation," IBM System Reference Library No GA22-6821-7 (September 1968).
- [Jo71] Johnston, J.B. "The Contour Model of Block Structured Processes," in [Wegn71] pp. 55-82.

- [Kn68] Knuth, D. The Art of Computer Programming, Vol. 1, Addison-Wesley, New York (1968).
- [La68] Lampson, B.W. "A Scheduling Philosophy for Multiprocessing Systems," Comm. ACM Vol. 11, No. 5 (May 1968), pp. 347-360.
- [Lan64] Landen, P.J. "The Mechanical Evaluation of Expressions," The Computer Journal, (January 1964), pp. 308-320.
- [Lan65] Landen, P.J. "A Correspondence between ALGOL 60 and Church's Lambda-Notation, Parts I and II," Comm. ACM, Vol. 8, Nos. 2 and 3 (February and March 1965), pp. 89-101, 158-165.
- [Lan66] Landen, P.J. "The Next 700 Programming Languages," Comm. ACM, Vol. 9, No. 3 (March 1966), pp. 157-164.
- [Lea69] Leavenworth, B.M. "The Definition of Control Structures in McG 360," Report RC 2376, IBM T.J. Watson Research Center, Yorktown Heights, New York, 1969.
- [Lu68a] Lucas, P. et al. Informal Introduction to the Abstract Syntax and Interpretation of PL/I, TR 25.083 IBM Laboratory, Vienna, Austria, June 1968.
- [Lu68b] Lucas, P. and Wolk, K. On the Formal Description of PL/I, Report of the IBM Laboratory, Vienna, Austria, December 1968.
- [Ma68] Madnick, S.E. "Multi-Processor Software Lockout," Proc. ACM National Conf. 1968, pp. 19-24.
- [McCar60] McCarthy, John "Recursive Functions of Symbolic Expressions and Their Computation by Machine," Comm. ACM, Vol 3, No. 4 (April 1960), pp. 184-195.
- [McCar66] McCarthy, John "A Formal Description of a Subset of Algol," in [Ste66] pp. 1-7.
- [McIl1] McIlroy, M.D. "Coroutines: Semantics in Search of a Syntax," unpublished, Oxford University.
- [Op65] Opler, A. "Procedure Oriented Language Statements to Facilitate Parallel Processing," Comm. ACM Vol. 8, No. 5 (May 1965), pp. 306-307.

- [Po71] Poupon, J. "Control Structure of PPL," in [Sch71].
- [Pr72] Prenner, C.J. et al. "An Implementation of Backtracking for Programming Languages," Proc. ACM National Conference 1972.
- [Ra68] Rappaport, R.L. "Implementing Multi-Process Primitives in a Multiplexed Computer System," Masters Thesis, MIT, November 1968.
- [Rey69] Reynolds, J.C. "A Set-Theoretic Approach to the Concept of Type," working paper, NATO Science Committee Conference, Techniques in Software Engineering, Rome, Italy, October, 1969.
- [Ro70] Roberts, L. and Wesler, B. "Computer Network Development to Achieve Resource Sharing," Proc. SJCC Vol. 36 (1970), pp. 543-549.
- [Sa66] Saltzer, J.H. "Traffic Control in a Multiplexed Computer System," Doctoral Dissertation, MIT, June, 1966.
- [Sa71] Saul, H. and Riddle, W. "Communicating Semaphores," Computer Science Department, Stanford University, STAN-CS-71-202 (February 1971).
- [Sch71] Schumann, S.(editor) "Proc. of the International Symposium on Extensible Languages," in SIGPLAN Notices, Vol. 6, No. 12, December 1971.
- [St68] Standish, T.A. A Preliminary Sketch of a Polymorphic Programming Language, Centro de Cálculo Electrónico, Universidad Nacional de México, June 1968.
- [St69] Standish, T.A. "Some Features of PPL, A Polymorphic Programming Language," in [Chris69] pp.20-26.
- [Ste66] Steel, T.B.(editor) Formal Language Description Languages for Computer Programming, North-Holland, Amsterdam, 1966.
- [Ta71] Taft, E. "FPL Users Manual," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1972
- [Tho71] Thomas, R.H. A Model for Process Representation and Synthesis, Doctoral Dissertation, MIT 1971.

- [vanW66] vanWijngaarden, A. "Recursive Definition of Syntax and Semantics," in [Ste66] pp. 13-18.
- [vanW69] vanWijngaarden, A. et al. Report on the Algorithmic Language Algol 68, MR 101, Mathematisch Centrum, Amsterdam, February 1969.
- [Weg70] Wegbreit, B. Studies in Extensible Languages, Doctoral Dissertation, Harvard University, Cambridge, Massachusetts, May 1970.
- [Weg71] Wegbreit, B. "The Treatment of Data Types in EL1," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1971.
- [Weg71a] Wegbreit, B. "Compactifying Garbage Collection in the Heap," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, 1971.
- [Weg72] Wegbreit, B. et al. "ECL Programmer's Manual," Center for Research in Computing Technology, Harvard University, Cambridge, Massachusetts, January 1972.
- [We67] Weissman, C. Lisp 1.5 Primer, Dickenson, Belmont, California, 1967.
- [Wegn69] Wegner, P. "Theories of Semantics," Technical Report No. 69-10, Center for Information Sciences, Brown University, September 1969.
- [Wegn71] Wegner, P. and Tou, J.L.(editors) "Proceedings of a Symposium on Data Structures in Programming Languages," in SIGPLAN Notices, Vol. 6, No. 2, February 1971.
- [Wi66] Wirth, N. "A Note on Program Structures for Parallel Processing," Comm. ACM, Vol. 9, No. 5 (May 1966), pp. 320-321.
- [Wi69] Wirth, N. "On Multi-Programming, Machine Coding, and Computer Organization," Comm. ACM Vol. 12, No. 9 (September 1969), pp. 489-498.

Unclassified

Security Classification

DOCUMENT CONTROL DATA - R & D

(Security classification of title, body of abstract and indexing annotation must be entered when the overall report is classified)

1. ORIGINATING ACTIVITY (Corporate author) Harvard University Cambridge, Massachusetts 02138		2a. REPORT SECURITY CLASSIFICATION UNCLASSIFIED	
		2b. GROUP N/A	
3. REPORT TITLE MULTI-PATH CONTROL STRUCTURES FOR PROGRAMMING LANGUAGES			
4. DESCRIPTIVE NOTES (Type of report and inclusive dates) None			
5. AUTHOR(S) (First name, middle initial, last name) Charles J. Prenner			
6. REPORT DATE August 1972		7a. TOTAL NO. OF PAGES 380	7b. NO. OF REFS 64
8a. CONTRACT OR GRANT NO. FI9628-71-C-0173		9a. ORIGINATOR'S REPORT NUMBER(S) ESD-TR-72-308	
b. PROJECT NO. c. 2801 d.		9b. OTHER REPORT NO(S) (Any other numbers that may be assigned this report)	
10. DISTRIBUTION STATEMENT Approved for public release; distribution unlimited.			
11. SUPPLEMENTARY NOTES		12. SPONSORING MILITARY ACTIVITY Deputy for Command and Management Systems Hq Electronic Systems Division (AFSC) L G Hanscom Field, Bedford, Mass. 01730	
13. ABSTRACT This dissertation applies the techniques of extensible languages to the problem of introducing multi-path control structures into programming languages. A control extension facility is defined which consists of a set of control primitives and a framework for combining them. The primitives are embedded in an existing extensible language--E11. Using the facility, it is possible to realize both conventional and non-conventional control regimes by extension. Such extensions are simplified through the use of the <u>control interpreter</u> , which allows the programmer direct control over the assignment of processors to paths. A set of examples is presented which demonstrates the power of the facility for both the implementation and clarification of complex control structures. Although the use of the primitives in the synthesis of control structures is emphasized, the primitives are also given a formal semantic definition which is used to demonstrate that they are feasible (i.e., they can be implemented on contemporary hardware) and that they have an efficient realization.			

Unclassified

Security Classification

14. KEY WORDS	LINK A		LINK B		LINK C	
	ROLE	WT	ROLE	WT	ROLE	WT
Programming languages Extensible programming languages Control structures Extensible control structures Definition mechanism Extension facility Formal semantic specification Interrupts Cooperating sequential processes Multiprogramming Multiprocessing Parallel processes Coroutines						

Unclassified

Security Classification